Coletânea Front-end

Uma antologia da comunidade front-end brasileira

ALMIR FILHO
BERNARD DE LUNA
CAIO GONDIM
DEIVID MARQUES
DIEGO EIS
EDUARDO SHIOTA
GIOVANNI KEPPELEN
LUIZ CORTE REAL
JAYDSON GOMES
REINALDO FERRAZ
SÉRGIO LOPES

Sumário

Capítulo 1

Uma coletânea com o melhor da comunidade front-end brasileira

A comunidade brasileira de programadores front-end e Web Designers é fantástica. Possui uma força imensa de elevar a qualidade da Web brasileira. Os milhares de representantes dessa comunidade produzem conteúdos ótimos em seus blogs, listas de discussão, Facebook e conversas de boteco. Nós nos encontramos em eventos memoráveis em todo o Brasil. Fazemos a diferença.

Esse livro quer ser uma pequena celebração desse sucesso.

Inspirados em projetos internacionais semelhantes – como o recomendadíssimo *Smashing Book* – trazemos essa **Coletânea de Front-end**. Um grupo de 11 autores de renome nacional na comunidade se juntou para escrever artigos que julgamos relevantes para a Web.

E mais importante até que o pequeno grupo de desbravadores desse projeto, queremos abrir o caminho para mais um canal da comunidade front-end. Queremos feedback, queremos que briguem conosco pra que lancemos novas edições, com mais

autores e mais temas.

1.1 Os capítulos e os autores

O livro está organizado em capítulos independentes, cada um com seu autor. Conversamos entre nós, mas cada tópico é expressão do próprio criador.

A única regra era: escrever algo memorável, que fizesse a diferença na Web brasileira. Nesse nosso meio que muda rápido, não queríamos falar da moda do mês. Focamos em temas atuais mas duradouros.

Começamos tentando rediscutir o papel da Web redescobrindo o **Progressive Enhancement** com *Luiz Real*. É impressionante como uma técnica com anos de idade é cada vez mais atual. E como há muita gente ainda ignorando essa prática essencial.

Na sequência, *Diego Eis* aborda **Responsive, Adaptive e Fault Tolerance** com muitas buzzwords e polêmicas. O autor mostra como a Web é genialmente imprevisível, adaptativa e flexível e discute o *mindset* correto pra lidar com isso.

Entra então *Bernard De Luna* falando sobre **Como criar frameworks CSS**. Mais que uma abordagem puramente técnica, é um tratado sobre a necessidade de padrões nos projetos e uma discussão interessante sobre o papel dos frameworks nas equipes Web.

Em seguida temos *Giovanni Keppelen*, que apresenta uma introdução detalhada ao **AngularJS**, um dos principais expoentes atuais do grupo de frameworks JavaScript MVC. Ele demonstra códigos práticos dos principais módulos do Angular e os motivos pelos quais você deve considerar essa ferramenta em suas aplicações.

A discussão sobre acessibilidade é bastante profunda com dois nomes de peso. Primeiro, *Reinaldo Ferraz* discute o coração da acessibilidade vista pelo W3C, mostrando as práticas fundamentais das **WCAG** que muitas vezes ainda são ignoradas no mercado.

Depois, *Deivid Marques* expande o tema abordando **WAI-ARIA** e as novas marcações de acessibilidade pensando em interações ricas na Web. Com exemplos excelentes, ele mostra como os novos atributos podem ser incorporados sem esforço nas suas aplicações.

Com a Web evoluindo a passos largos e browsers cada vez mais espertos, o que não faltam são novas APIs para explorar todo esse potencial. *Jaydson Gomes* mostra várias **APIs modernas** que você já pode usar hoje, como WebStorage, PostMessage, WebNotifications e History.

Como o tema é longo, *Almir Filho* mostra ainda mais APIs, agora com foco em acesso a **recursos de dispositivos modernos**. Aborda aspectos sobre câmera, acelerômetro, aúdio, GPS, vibração e mais. É o que você precisa saber pra dominar a Web em suas novas fronteiras.

Na sequência, *Caio Gondim* mostra como dominar todo esse mundo do front-end e facilitar o desenvolvimento com **Debug no browser e Dev Tools**. É um capítulo bastante aprofundado em como incorporar o uso das ferramentas de desenvolvimento no seu dia a dia com bastante produtividade e recursos úteis.

No último capítulo, *Eduardo Shiota* fala de **testes com JavaScript**. Um assunto de vital importância para garantir a qualidade de qualquer projeto que envolva front-end.

E, por fim, eu, *Sérgio Lopes*, idealizador da coletânea, escrevi esse prefácio e fiz o papel de editor, revisor técnico, cobrador de prazos e distribuidor de pitacos no texto alheio.

Em nome de todos os autores, espero que goste dos temas que escolhemos e aproveite o livro. Vamos mudar a Web juntos!

Sobre o editor

Sérgio Lopes é instrutor e desenvolvedor na Caelum, onde dá aulas de front-end e outras coisas. Escreve bastante conteúdo sobre front em seu blog (sergiolopes.org), twitter (@sergio_caelum) e outros lugares. Participa de muitos eventos no Brasil todo e publicou o livro *"A Web Mobile*" também pela editora Casa do Código.

Capítulo 2

Progressive Enhancement: construindo um site melhor para todos?

Com navegadores cada vez mais modernos, cheios de recursos, a tendência é que nossos sites também fiquem cada vez mais sofisticados. Porém, não podemos esquecer: nem todo mundo que acessa nossos sites está usando um navegador com os últimos recursos.

O primeiro pensamento que aparece na cabeça de muitos quando ouvem algo parecido é: mas eu posso obrigar meus usuários a atualizarem seus navegadores. Ou então: usuário com navegador desatualizado não merece acessar meu site! Mas será que sempre podemos exigir navegadores atualizados dos nossos usuários? E será que navegadores desatualizados são o único problema?

Quando fazemos um site, normalmente o fazemos para um público imenso de pessoas. Não podemos esquecer que, nesse público imenso, temos pessoas que não

gostam de JavaScript sendo executado em suas máquinas, pessoas que **não podem** atualizar seus navegadores, pessoas acessando a internet a partir de dispositivos limitados e pessoas com dificuldades motoras, visuais e auditivas que nem sempre conseguem utilizar o mouse para navegar ou dependem de leitores de tela para terem acesso ao nosso conteúdo. Mais ainda: temos um tipo de usuário muito importante para considerar, que não tem JavaScript nem CSS habilitados: as ferramentas de busca.

Isso significa, então, que não podemos fazer um site moderno e acessível ao mesmo tempo? Seria absurdo se a tecnologia evoluísse dessa forma. É por isso que os grupos que controlam o desenvolvimento das tecnologias que usamos na internet, como o W3C, têm sempre em mente a preocupação com esses casos citados. Mas como podemos desenvolver sites levando em conta esses cenários?

Graceful degradation

Uma primeira forma de pensar é desenvolver seu site primeiro para o público geral, que tem acesso aos navegadores mais modernos e não tem restrições de acessibilidade para, em um segundo momento, procurar atender aos usuários com mais limitações. Essa forma de pensar está presente há muitos anos no desenvolvimento de sistemas e a ideia de criar sistemas tolerantes a falhas; no mundo de desenvolvimento *front-end*, essa prática ficou mais conhecida como *graceful degradation*.

No entanto, pensar dessa forma pode nos levar a alguns problemas. Vamos pegar um exemplo para analisar: um botão de comprar em uma loja virtual. Você implementou a compra usando AJAX, para dar mais dinamicidade à navegação do usuário. Assim, no seu HTML, você provavelmente tem:

```
<input type="hidden" name="produto" value="123456">
<input type="number" name="quantidade">
<a href="#" id="comprar"><img src="icone-comprar.png"></a>
```

E, no seu JavaScript (usando jQuery, para encurtar um pouco):

```
$("#comprar").click(function() {
    var dadosCompra = {
        produto: $("[name=produto]").val(),
        quantidade: $("[name=quantidade]").val()
    };

    // enviamos os dados no formato JSON para o
```

```
    // servidor.
    // atualizaPagina é uma função que vai ser
    // executada depois que o servidor confirmar
    // a compra.
    $.post("/compra", dadosCompra, atualizaPagina, "json");
});
```

Isso resolve nosso problema para a maior parte dos usuários. Agora paramos para pensar: a quais casos não atendemos? Uma primeira possibilidade é pensar na acessibilidade da página: será que usuários com deficiência visual conseguirão comprar na nossa loja? Dificilmente. Afinal, nosso botão de comprar é uma imagem! O leitor de tela não vai conseguir ler o texto "comprar" a partir dela. Além disso, esse botão, por ser um link, pode gerar confusão nesses usuários. Podemos melhorar esta situação com um HTML mais semântico:

```
<form>
    <input type="hidden" name="produto" value="123456">
    <input type="number" name="quantidade">
    <button type="submit" id="comprar">
        <img src="icone-comprar.png" alt="Comprar">
    </button>
</form>
```

Repare no uso da tag `form` para indicar que estes controles, juntos, representam informações que serão enviadas para um servidor. Temos também a tag `button` com o tipo `submit` declarado, para indicar que essa imagem é um botão e que, ao ser clicado, enviará o formulário.

Agora, fica mais fácil de um leitor de tela descrever o que esse pedaço da tela faz, e nem precisamos alterar nosso JavaScript! Mas será que há algum outro caso que precisamos tratar? Sim, usuários sem JavaScript: como está a experiência atual para eles? Eles conseguem comprar no nosso site? O que acontece quando eles clicam no botão de compra? Absolutamente nada!

Repare que, tendo a preocupação com *graceful degradation*, precisamos lembrar de todos os cenários que deixamos de lado ao desenvolver nosso site com as últimas tecnologias. E, se precisamos lembrar desses cenários, qual a chance de esquecermos algum? Mais ainda: agora precisamos implementar uma solução sem JavaScript. Será que é possível? Nosso servidor nos devolve um JSON como resultado, não uma página. Para mostrar algo útil para o usuário, **precisamos** de JavaScript, agora. Ou seja, por termos desenvolvido uma solução sem pensar nos casos mais limitados,

acabamos caindo em um beco sem saída. Precisaremos **refazer boa parte da nossa solução**, inclusive do lado do servidor.

Como fazer, então, para não corrermos o risco de esquecermos esses cenários mais limitados? **Começando por eles**; essa é a ideia do ***progressive enhancement***.

Progressive Enhancement

Para entender a diferença entre os dois, vamos usar o mesmo cenário: precisamos implementar o botão para comprar um produto em uma loja virtual. Porém, vamos começar por um cenário bem limitado: um navegador baseado em texto (o navegador Lynx, por exemplo, http://lynx.browser.org/) . Nesse tipo de navegador, a única ferramenta que temos disponível é o HTML. Como implementar um botão de compra usando apenas HTML? Com algo próximo do que já tínhamos:

```
<form action="/comprar" id="comprar">
    <input type="hidden" name="produto" value="123456">
    <input type="number" name="quantidade">
    <button type="submit">Comprar</button>
</form>
```

Repare em uma diferença importante: o atributo `action` no formulário. Com ele, o navegador sabe para qual endereço no servidor os dados deste formulário devem ser enviados; não precisamos de JavaScript nenhum para ensinar isso para o navegador. Repare, também, em uma outra diferença: colocamos o texto "Comprar" dentro do botão em vez da imagem. Até poderíamos deixar a imagem mas, por ser uma questão estética, podemos deixar para um segundo momento, quando estivermos pensando no CSS dessa página. Por fim, vale observar que essa decisão de começar pelo cenário mais limitado influencia também o lado servidor da aplicação: ao comprar, vamos enviar nossos dados no formato padrão do navegador e não no formato JSON.

Agora podemos passar para um cenário um pouco mais complexo: usuários com deficiência visual. Com eles, já podemos usar JavaScript. Para implementar a compra com AJAX, como queríamos inicialmente, não teremos quase nenhum código agora:

```
$("#comprar").submit(function() {
    $.post(this.action, $(this).serialize());
});
```

Veja que, agora, não precisamos mais nos preocupar com os dados do formulário individualmente nem com a URI da compra. Por estarmos usando um formulário

semântico, podemos simplesmente pedir para o jQuery pegar os dados desse formulário e enviá-lo como o navegador faria, porém de forma assíncrona.

Nesse mesmo cenário, também poderíamos começar a pensar no design da página, usando elementos grandes e/ou com bom contraste. E, ao testar esse cenário com um leitor de tela, provavelmente lembraremos de colocar marcações da especificação WAI-ARIA, para melhorar a usabilidade da página.

Quando começamos por um cenário mais limitado, é natural querermos solucioná-lo adequadamente. Isso nos força a pensar e desenvolver de uma forma que favorece um HTML mais semântico e desacoplado de CSS e JavaScript. Ganhamos não apenas um site que funciona bem para todos; ganhamos também um código **mais limpo e fácil de manter**.

No entanto, aplicar o *progressive enhancement* não é tão fácil como pode parecer à primeira vista. Diversas questões aparecem: qual vai ser meu cenário mais limitado? Por onde começar? Como acrescentar funcionalidade sem quebrar o que eu já tinha? Nas próximas seções, vamos analisar essas questões e como elas se aplicam a cada uma das tecnologias que usamos para desenvolver nossas páginas.

2.1 Por onde começar?

Quando começamos a pensar seguindo a linha do *progressive enhancement*, vemos que ele influencia os mais diversos aspectos do nosso projeto.

Por exemplo, se formos desenvolver um site para divulgar um produto, podemos pensar, antes de mais nada: qual o nosso público-alvo? Será que precisamos nos preocupar com navegadores antigos? Qual parcela de visitantes do meu site virá de dispositivos móveis? Quão importante é a integração com redes sociais? O que posso oferecer para meus visitantes com configurações mais limitadas?

Repare que essas questões são muito mais relacionadas ao seu negócio do que a questões técnicas. Saber respondê-las é muito importante para aplicarmos os conceitos de *progressive enhancement* corretamente, desde o começo do projeto.

Uma vez que se tenha claro o público-alvo, normalmente começam duas frentes de trabalho: o desenvolvimento da lógica do site (*back-end*) e o projeto da interface. No desenvolvimento do *back-end*, saber qual o público-alvo vai influenciar em quais funcionalidades serão implementadas e como. No desenvolvimento da interface, o paradigma do *progressive enhancement* influenciará o fluxo desse desenvolvimento, como veremos mais adiante.

Ou seja, o *progressive enhancement* não é apenas uma forma de desenvolver o

código do *front-end*: **é uma forma diferente de pensar o desenvolvimento do produto como um todo**. Sendo assim, uma possível resposta para a pergunta "por onde começar?" é: pelo planejamento do seu produto. Uma vez que se tenham bem claros os objetivos do produto, as decisões técnicas (quais navegadores suportar, quais tecnologias usar etc.) tornam-se simples.

Infelizmente, isso dificilmente acontece em um cenário real. Normalmente temos que lidar com um público-alvo pouco conhecido, dificuldades na hora de tomar decisões de produto mais técnicas junto ao cliente e limitações no orçamento e no prazo de entrega. Ainda assim, é possível aplicar o *progressive enhancement*. Se não sabemos qual o cenário mais limitado a que vamos atender, podemos começar pelo mais limitado. Se não temos orçamento e/ou prazo suficientes para desenvolver todas as funcionalidades desejadas, podemos entregar as que atendem aos cenários mais limitados primeiro. Se o cliente não sabe quais tecnologias os visitantes do site vão usar para acessar o conteúdo, começamos com o mínimo possível de tecnologias.

Entregando com qualidade

Em todo projeto, temos três variáveis possíveis de serem controladas: orçamento, prazo e qualidade. No entanto, é impossível controlar as três **ao mesmo tempo**: se fixamos prazo e qualidade, o produto sai mais caro; se fixamos prazo e orçamento, a qualidade diminui; se fixamos orçamento e qualidade, o desenvolvimento demorará mais.

Contudo, sacrificar a qualidade normalmente não é uma boa alternativa. Sendo assim, se o prazo não é suficiente para entregar todas as funcionalidades, não entregue nada mal feito; entregue o que for possível. E, se você estiver seguindo o *progressive enhancement*, as funcionalidades que você entregar são aquelas que influenciam os cenários mais limitados, ou seja, o maior número de usuários possível.

Repare que atender ao cenário mais limitado primeiro não significa necessariamente começar a desenvolver pelos navegadores mais antigos (Internet Explorer 6, estou olhando para você) ou para navegadores baseados em texto. Novamente, **é uma decisão que cabe ao cliente do projeto**. O papel que cabe a nós, desenvolvedores, é informar o cliente das consequências de incluir ou excluir determinadas tecnologias do projeto.

Vale lembrar, também, que, cada vez mais, podemos fazer mais com menos.

Os navegadores vêm oferecendo ferramentas cada vez mais avançadas para nós, de forma que não dependemos mais de tecnologias como Flash ou Java para resolver a maior parte dos problemas. Além disso, o CSS e o HTML vêm ganhando recursos que nos livram da necessidade de usar JavaScript em diversos casos. Um exemplo clássico é a implementação de navegação em abas: antigamente, esse tipo de solução precisava de JavaScript para ser implementada; hoje em dia, é possível implementá-la apenas com CSS e HTML, tornando-a disponível para um público ainda maior.

2.2 Progressive Enhancement e HTML

Se queremos começar do cenário mais restrito, precisamos pensar em quais tecnologias podemos usar nesse cenário. Há várias possibilidades mas, sem dúvida, todas elas têm suporte ao HTML.

O HTML é a base de toda página Internet e, portanto, todo usuário do nosso site, seja humano ou máquina, tem que entender pelo menos HTML; sempre podemos contar com ele.

No entanto, precisamos lembrar que o HTML passou por diversas modificações desde que foi criado, ou seja, temos várias versões de HTML. Qual delas vamos suportar? É aí que a história começa a complicar.

Atualmente, todos os navegadores, em sua última versão, trabalham com a versão 5 do HTML. Porém, o que isso significa? Quando dizemos *HTML 5*, na verdade, estamos falando de uma série de novas funcionalidades agregadas à linguagem HTML: novas tags, novos atributos, novos controles de formulário, novas APIs, além de modificações de sintaxe e semântica. São muitos novos recursos, de modo que os navegadores, em sua maior parte, **ainda não implementam toda a especificação do HTML5**.

Ora, então não podemos usar o HTML 5? Felizmente, a especificação da linguagem HTML já foi feita pensando em *progressive enhancement*. Para entender o porquê, vamos analisar o que acontece quando um navegador analisa o seguinte HTML:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Teste de HTML5</title>
    </head>
```

```
    <body>
        <header>
            <h1>Página com HTML5</h1>
            <span>Bacana, né?</span>
        </header>
        <section>
            <h1>Hora atual</h1>
            <time>22:39</time>
            <p>Um controle de form novo:</p>
            <input type="range" min="1" max="10">
        </section>
    </body>
</html>
```

Se o navegador já implementa suporte a todo esse código, não há muito segredo: logo no começo do código, estamos usando a declaração `DOCTYPE` para indicar que estamos usando a **última versão do HTML**, ou seja, o navegador deve mostrar todo o seu potencial! Assim, temos como resultado uma página exatamente como esperávamos: um cabeçalho com o título da página e uma seção de texto com o título dela, uma hora, um parágrafo e um controle do tipo *slider*.

E se o navegador não implementa suporte a todos esses recursos, o que acontece? Já na versão **4**, a especificação do HTML recomenda que [1]:

- Se um agente de usuário encontrar um elemento que não reconhece, ele deve tentar renderizar seu conteúdo;
- Se um agente de usuário encontrar um atributo que não reconhece, ele deve ignorar a declaração completa deste atributo, isto é, o atributo e seu valor;
- Se um agente de usuário encontrar um valor de atributo que não reconhece, ele deve usar o valor padrão para aquele atributo.

Repare que a recomendação é bem favorável à aplicação do *progressive enhancement*: mesmo que o navegador não entenda exatamente o que estamos querendo dizer, ele vai mostrar o conteúdo para o usuário; **podemos incrementar a semântica e a interatividade do nosso HTML sem quebrar os navegadores mais limitados!**

No exemplo dado anteriormente, se o navegador não suporta a tag `<time>` e o `input` do tipo `range`, ainda assim o usuário verá a hora e um controle de formulário (uma caixa de texto, que é o controle de formulário padrão).

Porém, por ser apenas **uma recomendação**, nem todos os navegadores seguem essas diretrizes e/ou seguem-nas de modos *bem diferentes*. Em particular, o uso de tags do HTML 5 no Internet Explorer nas versões anteriores à 9 traz problemas de estilização e interpretação da estrutura da página [2]. A versão 5 da especificação é bem mais clara quanto ao tratamento de erros, o que deve melhorar a vida do desenvolvedor *front-end* em um futuro não muito distante.

Suportando HTML 5 em versões mais antigas do Internet Explorer

Para contornar os problemas decorrentes do uso de tags do HTML 5 nas versões mais antigas do Internet Explorer, foi criada uma pequena biblioteca escrita em JavaScript capaz de habilitar um suporte mínimo às novas tags: o html5shiv (https://code.google.com/p/html5shiv/) . Ela resolve os problemas de estilização e interpretação da estrutura da página mas não adiciona funcionalidade ao navegador. Ou seja, se você quer fazer com que o Internet Explorer suporte o `<input type="datetime">`, por exemplo, essa biblioteca **não resolve o problema**. Veremos mais adiante uma possível resolução para esse caso.

Além disso, vale notar que, por ser uma solução baseada em JavaScript, ela não atende 100% dos usuários do site: aqueles que desabilitaram a execução de JavaScript verão a página com os problemas de estilização originais.

2.3 Progressive Enhancement e CSS

Uma vez que tenhamos a base – o HTML – de uma página bem definida, podemos começar a implementar o design com CSS. Vale observar que o processo de design, por si só, também pode seguir o *progressive enhancement*, mas nosso foco aqui vai ser nas implicações do *progressive enhancement* no uso do CSS.

Assim como o HTML, o CSS é uma tecnologia já antiga e que passou por uma grande evolução, culminando no que chamamos agora de **CSS 3**. E, assim como no caso do HTML, os navegadores não implementam suporte a todas as novidades, mesmo porque elas continuam vindo (grids, formas, regiões para citar alguns exemplos), mas os navegadores ignoram as propriedades desconhecidas, de modo que

também é muito fácil ir incrementando nosso visual de acordo com as funcionalidades disponíveis!

Além disso, o CSS tem uma outra característica peculiar que facilita o *progressive enhancement*: quando uma propriedade aparece duplicada, apenas a última declaração é considerada. Para entender por que isso nos ajuda, vamos considerar o seguinte exemplo: uma foto com a legenda sobreposta a ela.

Uma forma de descrever esse cenário semanticamente com HTML é:

```
<figure>
    <img src="..." alt="Ancoradouro em Belize">
    <figcaption>Passe as férias no Caribe!</figcaption>
</figure>
```

Para posicionar a legenda em cima da foto, podemos dar à `<figure>` um posicionamento relativo e, à legenda, um posicionamento absoluto:

```
figure {
    position: relative;
}
figcaption {
    position: absolute;
    bottom: 0;
}
```

Com isso, basta definir a cor de fundo. Note que a cor é um preto translúcido. Até recentemente, não havia uma forma de fazer essa cor de fundo apenas com CSS, era necessário criar uma imagem com essa cor e usá-la de fundo, o que não é muito

flexível nem performático. Então, para os navegadores mais limitados, vamos usar uma cor opaca mesmo:

```
figcaption {
    background-color: black;
    color: white;
}
```

Uma vez implementado o layout para cenários mais limitados, podemos ir embelezando nossa página com as novidades do CSS 3. No nosso caso, agora podemos colocar a cor translúcida de fundo. Porém, se substituirmos a nossa declaração atual, quebraremos nosso trabalho anterior e deixaremos um texto branco diretamente em cima de uma foto, o que não é nada usável. Então vamos aproveitar o comportamento do CSS de sobrescrever declarações anteriores para suportar tanto navegadores mais antigos como mais novos:

```
figcaption {
    background-color: black;
    background-color: rgba(0, 0, 0, 0.8);
    color: white;
}
```

Os navegadores mais antigos só entenderão a primeira declaração de `background-color`; os mais novos entenderão as duas e usarão apenas a última. Veja como esse comportamento nos permite fazer *progressive enhancement* de um jeito fácil: basta ir acrescentando as funcionalidades mais recentes **abaixo** das mais antigas!

Indo além com CSS

Quando pensamos em *progressive enhancement*, devemos pensar em dar a melhor experiência possível para os cenários limitados. Isso implica em tirar o maior proveito possível das ferramentas que temos nesses cenários.

A maior parte dos sites que vamos desenvolver no nosso dia a dia precisará de CSS para ser visualmente agradável e atrair mais usuários. Ou seja, mesmo nos cenários mais limitados, já estaremos dependentes de CSS. Já que é assim, então podemos explorá-lo o máximo possível!

Muitos requisitos de *front-end* que normalmente implementamos com JavaScript podem ser feitos apenas com CSS, ou seja, sem depender de *mais uma* tecnologia.

Um exemplo clássico de requisito deste tipo é a navegação em abas. Uma possível representação para abas no HTML é:

```
<ul class="abas">
    <li class="aba" id="aba-1">
        <a href="#aba-1">Sobre</a>
        <section class="conteudo">
            ...
        </section>
    </li>
    <li class="aba" id="aba-2">
        <a href="#aba-2">Descrição</a>
        <section class="conteudo">
            ...
        </section>
    </li>
    <li class="aba" id="aba-3">
        <a href="#aba-3">Fotos</a>
        <section class="conteudo">
            ...
        </section>
    </li>
</ul>
```

Com base nesse HTML, podemos criar abas usando a pseudoclasse `:target` e alguns truques de posicionamento:

```
.abas {
    position: relative;
}
.aba {
    display: inline;
}
.aba > a {
    float: left;
    padding: 0.5em 1em;
    background: linear-gradient(#FFF, #EEE);
    border-width: 1px;
    border-style: solid;
    border-color: #CCC #CCC #FFF;
}
.aba:not(:target) a {
```

```
    border-bottom: 0 none;
}
.aba:target a {
    background: white;
}
.conteudo {
    position: absolute;
    left: 0;
    top: calc(2em + 4px); /* altura do link + bordas */
    z-index: -2;
    border: 1px solid #CCC;
    background-color: white;
}
.aba:target .conteudo {
    z-index: -1;
}
```

Essa e outras implementações você pode conferir com mais detalhes em http://css-tricks.com/examples/CSSTabs.

Vale notar que, às vezes, passamos a depender de funcionalidades do CSS que excluem mais usuários do que a dependência de JavaScript. Nesta implementação, por exemplo, estamos excluindo todas as versões do Internet Explorer anteriores à 9. É necessário, portanto, analisar qual implementação atende mais usuários com menos recursos.

2.4 Progressive Enhancement e JavaScript

Desenvolver pensando primeiro nos cenários mais limitados já evita que caiamos no tipo de armadilha mostrada na introdução deste capítulo, com o formulário AJAX. No entanto, quando adicionamos JavaScript à página, precisamos tomar certos cuidados para não quebrar o trabalho já feito, assim como no CSS.

Da mesma forma que devemos pensar no CSS como algo **a mais** em uma página, devemos também pensar no JavaScript dessa forma. Isso significa que, na medida do possível, o código JavaScript **não deve interferir** no seu HTML. Por exemplo, em vez de fazer um link virar uma ação em JavaScript com

```
<a href="#" onclick="maisProdutos()">Mais produtos</a>
```

devemos preservar o HTML original

```
<a href="mais-produtos.html">Mais produtos</a>
```

e adicionar a funcionalidade JavaScript usando o próprio JavaScript, em algo como

```
document.querySelector('[href="mais-produtos.html"]')
    .addEventListener('click', maisProdutos);
```

Dessa forma, nosso site continua funcionando perfeitamente, mesmo que o JavaScript apresente algum problema, e **essa é uma das principais vantagens do *progressive enhancement* para o seu desenvolvimento**. Esse tipo de pensamento é conhecido entre os desenvolvedores JavaScript como *JavaScript não-obstrutivo*.

É fato que, em muitos casos, algumas funcionalidades só estarão presentes para usuários com JavaScript habilitado. Por exemplo: imagine que queremos fazer uma caixa de busca para filtrar uma lista de resultados, como na figura a seguir. Como podemos desenvolver algo deste tipo pensando em *progressive enhancement*?

A resposta para essa pergunta sempre é **começando pelo cenário mais limitado**. E o que podemos oferecer para nossos usuários só com HTML, por exemplo? Uma simples lista:

```
<h1>Sobremesas</h1>
<ul class="resultados">
    <li><a href="receita?id=123">Bolo de nozes</a></li>
    <li><a href="receita?id=234">Estrogonofe de nozes</a></li>
    <li><a href="receita?id=345">Bolo de chocolate</a></li>
    <li><a href="receita?id=456">Torta de chocolate</a></li>
    <li><a href="receita?id=567">Torta de maçã</a></li>
</ul>
```

Depois de **incrementar** a página com o estilo, passamos à funcionalidade da busca. E, para implementá-la, podemos facilmente pensar em **mudar nosso HTML** para algo do tipo:

```
<h1>Sobremesas</h1>
<input type="search" onkeyup="filtra()">
<ul class="resultados">
    <li><a href="receita?id=123">Bolo de nozes</a></li>
    <li><a href="receita?id=234">Estrogonofe de nozes</a></li>
    <li><a href="receita?id=345">Bolo de chocolate</a></li>
    <li><a href="receita?id=456">Torta de chocolate</a></li>
    <li><a href="receita?id=567">Torta de maçã</a></li>
</ul>
```

Porém, ao fazermos isso, sem percebermos, estamos oferecendo uma página "quebrada" para os usuários sem JavaScript. Afinal, para que serve esta caixa de busca para eles? **Nada**; portanto, ela simplesmente **não deveria estar lá**. Mas como fazemos para ela aparecer só para os usuários que têm JavaScript funcionando? **Inserindo-a via JavaScript**!

```
var busca = document.createElement('input');
busca.type = 'search';

var resultados = document.querySelector('.resultados');
resultados.parentNode.insertBefore(busca, resultados);
```

Se você estiver usando jQuery, dá para fazer isso em **uma linha**:

```
$('<input type="search">').insertBefore('.resultados');
```

Simples, não? E, de quebra, ganhamos uma organização de código melhor: o que é relacionado a JavaScript **fica no JavaScript**. Assim não poluímos nosso HTML.

Ou seja, quando temos a preocupação do *progressive enhancement* na implementação das funcionalidades que dependem de JavaScript, uma boa prática é usar **apenas** JavaScript para acrescentá-las ao site. Com isso, asseguramos que não quebraremos o trabalho anterior e, de quebra, deixamos nosso código mais organizado.

Lidando com JavaScript limitado

Um ponto que não gera dores de cabeça no HTML e no CSS mas que, no JavaScript, é bastante complicado é lidar com funcionalidades faltantes. Vimos que, com relação ao HTML, o navegador mostra informações de tags desconhecidas e, com relação ao CSS, o navegador ignora propriedades e valores não suportados; o mesmo não acontece com o JavaScript: qualquer comando que não seja suportado pelo navegador gerará um **erro de JavaScript**, consequentemente parando toda a execução do código.

Lidar com as limitações e diferenças entre os navegadores pode ser bastante trabalhoso. Por exemplo, para selecionar elementos da página, podemos usar a função `document.querySelector`, como fizemos no exemplo anterior. No entanto, essa função não está presente em todos os navegadores. Para que nosso código funcione em todos os navegadores, podemos usar a função `document.getElementsByClassName`:

```
var resultados = document.getElementsByClassName('resultados')[0];
```

Neste caso, o código ficou apenas um pouco mais complicado do que antes. Mas será que para o próximo exemplo seria simples oferecer uma alternativa?

```
var linksExternos = document.querySelectorAll('a[href^=http]');
```

Felizmente, temos atualmente no mercado diversas bibliotecas que cuidam dessas diferenças e limitações para nós, tais como jQuery, Prototype, MooTools e Dojo. De um modo geral, essas bibliotecas nos permitem escrever código JavaScript sem precisarmos nos preocupar tanto com os cenários mais limitados. Vale a pena usá-las, principalmente se você precisa dar suporte a navegadores mais antigos.

Apesar de abstrairem problemas de compatibilidade entre os navegadores, essas bibliotecas não nos livram de lidar com um outro problema: presença ou ausência de **funcionalidades** entre navegadores.

Imagine, por exemplo, que você está construindo um site de uma rede de restaurantes e quer mostrar para um usuários os pratos exclusivos de sua região em

destaque. Para isso, você pode perguntar em qual cidade o usuário está ou, mais interessante, tentar detectar onde ele está. Isso é possível por meio da API de geolocalização dos navegadores:

```
navigator.geolocation.getCurrentPosition(function(position) {
    // descobre a cidade baseado nas coordenadas
});
```

Porém, nem todo navegador tem suporte a essa API e, quando não há suporte, o objeto `geolocation` não está definido e, portanto, o código anterior gera um erro.

Uma primeira solução seria verificar o navegador do usuário para habilitar ou não este trecho de código. Porém, com a quantidade de navegadores que temos atualmente, é simples dizer quais suportam geolocalização?

Uma outra abordagem, mais simples e robusta, é verificar se o navegador do usuário **tem a funcionalidade desejada**, independente de qual seja. Neste nosso caso, é simples fazer isso: basta verificar se o objeto `geolocation` está definido antes de usá-lo.

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(function(position) {
        // descobre a cidade baseado nas coordenadas
    });
}
```

Nos casos em que a detecção da funcionalidade é mais complicada, podemos usar a biblioteca *Modernizr* (http://modernizr.com/) , cujo propósito é justamente detectar as funcionalidades disponíveis no navegador.

Programando desta forma, podemos garantir que funcionalidades mais avançadas não quebrarão nosso código, permitindo o *progressive enhancement* do lado do JavaScript.

Em alguns casos, ainda é possível estender a funcionalidade padrão do navegador com o uso de bibliotecas escritas em JavaScript que emulam alguns recursos. Podemos, por exemplo, prover uma implementação do objeto `geolocation` para os navegadores que não o possuem (a biblioteca *Webshims Lib* faz isso, por exemplo – https://github.com/aFarkas/webshim) . Bibliotecas com esse propósito – acrescentar uma funcionalidade a navegadores que ainda não a suportam – são conhecidas como ***polyfills***.

Vale observar que os *polyfills* não só permitem o uso de recursos avançados do JavaScript em navegadores mais simples, como também de recursos mais modernos

de HTML e CSS. Existem *polyfills* para os controles de formulário do tipo `date` e para a propriedade `box-sizing`, por exemplo.

Assim, dependendo de seu público alvo, *polyfills* podem ser uma alternativa interessante para melhorar a experiência dos usuários com mais limitações. Mas é importante lembrar que *polyfills* apenas **emulam** a funcionalidade original e, portanto, têm limitações e, por dependerem de JavaScript (e, em alguns casos, de outras tecnologias, como Flash), não agregam a funcionalidade desejada a todos os usuários. Novamente, desenvolver seu site pensando em *progressive enhancement* é essencial para que **todos** os usuários tenham acesso às informações contidas nele.

2.5 Quando o Progressive Enhancement não é uma alternativa

A abordagem do *progressive enhancement* resolve muitos problemas do desenvolvedor *front-end* ao forçar o foco primeiro na parte mais importante de um site, que é **prover o conteúdo**. No entanto, *progressive enhancement* tem suas desvantagens e nem sempre é aplicável.

Quando desenvolvemos pensando primeiro nos cenários mais limitados, conseguimos planejar nosso desenvolvimento de modo a tornar nosso site minimamente acessível nesses cenários. No entanto, isso pode ser restritivo para o processo criativo de desenvolvimento de um site. Imagine, por exemplo, que você precisa fazer uma página mostrando as funcionalidades de um aparelho celular novo. O jeito mais simples e que atende a todos os usuários é simplesmente montar uma lista dessas funcionalidades, possivelmente com imagens ilustrativas. Porém, pensando dessa forma, você pode acabar deixando de apresentá-las de uma forma mais criativa, como um menu interativo. Para não correr esse risco, vale a pena pensar primeiro em como queremos que nosso site fique no final para, daí, começar a implementar pelo cenário mais limitado. Essa ideia de projeto final, inclusive, pode servir de guia para soluções mais criativas mesmo nesses primeiros cenários.

Agora, em projetos em que há uma dependência muito grande de recursos modernos dos navegadores, como num jogo ou um site que trabalha com a webcam do usuário, o *progressive enhancement* acaba não trazendo vantagens e, por outro lado, pode dificultar o desenvolvimento. Nessas situações, é possível desenvolver uma versão mais simples, sem as funcionalidades principais, para os cenários mais limitados, usando *progressive enhancement*. Essa abordagem é seguida, por exemplo, pelo Gmail, o serviço de e-mail da Google. A versão principal do cliente web

é desenvolvida usando recursos avançados de JavaScript. Para simplificar o desenvolvimento dessa versão e ainda permitir o acesso aos e-mails nos navegadores mais limitados, foi desenvolvida uma versão baseada apenas em HTML.

Há, ainda, casos em que o *progressive enhancement* não traz muitas vantagens. Em um site feito para uma intranet, por exemplo, temos controle total de qual será o navegador do usuário e quais funcionalidades ele tem. Quando temos esse controle, acaba não sendo interessante fazer *progressive enhancement*. Porém, ainda assim, preocupar-se com a acessibilidade da página e desenvolver o CSS e o JavaScript de forma incremental, usando o HTML como base, traz vantagens e não deixa de ser importante.

Mesmo nos cenários em que *progressive enhancement* não é aplicável, é interessante ter em mente as preocupações dessa forma de desenvolvimento. Desenvolver para a web é desenvolver para todos, independente de plataforma, navegador, língua e capacidades, e essa é a principal preocupação do *progressive enhancement*: fazer da web uma só, para todos, como idealizou o criador da web, Tim Bernes-Lee [3].

[1] http://www.w3.org/TR/html401/appendix/notes.html#notes-invalid-docs [2] http://diveintohtml5.info/semantics.html#unknown-elements [3] http://news.bbc.co.uk/2/hi/technology/6983375.stm

Sobre o autor

Luiz é bacharel e mestre em Ciência da Computação e trabalha como desenvolvedor e instrutor na Caelum atualmente. Adora programar para fazer sistemas não só úteis, como bonitos por dentro e por fora, de preferência usando a Web como plataforma. Por isso mesmo, ama HTML, CSS e JavaScript. Colabora com projetos de código aberto como VRaptor, MedSquare e Tubaina, e também escreve para o blog VidaGeek.net.

Capítulo 3

Responsive, Adaptive e Fault Tolerance

3.1 Responsive é quebrado

Eu não quero ferir o sentimento de ninguém. Eu sei que o *Responsive Web Design* (RWD) é coisa linda, contudo, por favor, não o trate como o *Santo Graal.* O RWD não é a solução para todos os problemas. O Responsive é uma boa solução parcial de um problema muito maior do que apenas adequar websites para diversas telas.

Suposições

Tudo o que você sabe sobre o usuário não passa de suposições. Trabalhar com Web é difícil por causa disso. Você nunca tem certeza sobre como seu usuário pode acessar seu site. Você tem uma suposição, uma ligeira ideia de que ele pode acessar seu site com um determinado browser, em um determinado sistema, em um determinado dispositivo. Você nunca sabe nada com certeza absoluta.

Você pode falar que uma grande maioria dos seus usuários acessa o site usando uma determinada resolução. Mas quando vasculhamos os dados, esses números são fragmentados e é difícil descobrir pela quantidade de dispositivos e browsers que eles usam. Você pode descobrir que a maioria dos seus usuários usam IE, mas também que há uma grande parcela que usa Safari para Mobile. É tudo muito sensível, entende?

Estruturas sim, módulos depois

O Responsive pode adequar websites para vários tipos de tamanho de tela, mas essa adequação só chega até a estrutura do site. Você pode falar que o site terá 3, 2 ou 1 coluna e mais nada. As Media Queries ajudam a resolver nosso problema estrutural. Essa parte é super importante, mas não é suficiente. Há ainda outro grande problema que envolve cada módulo do site. Um elemento usado no desktop pode não ser o ideal nos celulares.

Entenda que não sou contra o RWD, pelo amor de Deus... Eu o amo de paixão. Mas quero que você pense friamente que ele, sozinho, não resolve todos os seus problemas. É aí que um conceito bem maior entra em cena, chamado **Adaptive Web Design**, que abrange uma série de soluções para adaptar seu website/produto a diversos cenários e contexto, onde o RWD é apenas uma peça do quebra cabeças. Você precisa considerar muitas coisas além de layouts flexíveis para que seu website seja de fato acessado por qualquer um.

Existem cenários que não foram bem explorados e que ainda vão nos atormentar por muito tempo, como é o caso dos navegadores em consoles. Pode ser que você descubra que seus usuários, por algum motivo qualquer, estejam acessando seu website via Xbox, Playstation etc. A ideia de adaptar websites para consoles é algo pouco explorado. Para começar, você não tem mouse, nem teclado. Você tem um joystick que não foi preparado para essa tarefa.

Muitos formatos estão surgindo de todos os cantos e isso é algo realmente perturbador. Smartwatchs serão a próxima onda e muito provavelmente os usuários sentirão a necessidade de acessar informações em websites usando um dispositivo assim. E aí?

3.2 Adaptive Delivery é quebrado

Já vou avisando que esse campo também é bem desconhecido. Ninguém tem as respostas corretas e ninguém sabe exatamente o que fazer até agora. A solução correta

é aquela que funcionou no seu projeto.

O *Adaptive Delivery* também é uma peça do quebra-cabeça, como o RWD. Ele se resume em: enviar partes do website ou o website inteiro adaptado para o contexto que o usuário se encontra. Se você está usando mobile, vou servir partes do código que se adegue ao mobile. Se for TV, vou servir partes do site que facilitam o uso em uma tela grande e assim por diante.

Como você faz isso? Depende. Tem gente que usa soluções *back-end*, identificando via os headers qual plataforma/contexto o usuário está e aí pede para o servidor para enviar determinado pedaço de código. Tem gente que resolve tudo no *front-end*, com JavaScript. Qual dos dois está errado? Depende.

Às vezes, uma simples troca de classe de um elemento já o faz mudar totalmente, resolvendo sua estrutura para diversos dispositivos. Às vezes, você precisa desconstruir todo o elemento, guardando se conteúdo, para construir o HTML de algo diferente.

O exemplo das tabs

Costumo usar um exemplo bem básico para você entender a brincadeira. As tabs (ou guias, como você preferir) são bem usadas em sistemas e websites. O problema com as tabs é que elas são bem usadas em Desktops e não nos mobiles. Quando uma tab quebra de linha, perde-se totalmente a analogia inicial.

Em uma tela menor, como ficaria:

Muito, muito, muito ruim.

O mais legal seria que no mobile esse elemento virasse um dropdown. O uso seria mais intuitivo, economizaríamos espaço e manteríamos decente a experiência do usuário:

Essa é uma solução básica e você pode nem gostar dela. Não há problema nisso. O ponto é que você entendeu o conceito: formas diferentes de entregar o conteúdo

dependendo do dispositivo.

Existem outros exemplos extremos de *Adaptive Delivery*, nos quais não mudamos apenas uma parte do código, mas o site inteiro, como é o caso da Lufthansa. Aliás, esse site da Lufthansa é um case clássico de *Adaptive Delivery*, em que entregamos o código exato de acordo com o contexto do usuário. Existem vários websites que explicam esse case. Dê uma pesquisada no Google sobre o site deles.

Existem dois conceitos que você já deve ter cansado de ouvir e estudar: *Fault Tolerance* e *Progressive Enhancement*, assunto inclusive do primeiro capítulo do livro. Os dois servem como base para produzir websites que se adequam de verdade a qualquer tipo de dispositivo, não importa qual a tela ou qual plataforma.

Se você levar a sério esses dois conceitos, você conseguirá fazer websites compatíveis com qualquer browser e principalmente com qualquer contexto que o usuários se encontrar.

3.3 O que realmente importa: Progressive Enhancement e o Fault Tolerance

Fault Tolerance é como as máquinas tratam um erro quando ele acontece. É a habilidade do sistema de continuar em operação quando uma falha inesperada ocorre. Isso acontece a todo momento com seu cérebro. O sistema não pode parar até que esse erro seja resolvido, logo, ele dá um jeito para que esse erro não afete todo o resto. A natureza inteira trabalha dessa forma. Os browsers trabalham dessa forma. É por isso que você consegue testar as coisas maravilhosas do CSS3 e do HTML5 sem se preocupar com browsers antigos.

Já temos as vantagens do Fault Tolerance desde o início

Por exemplo, quando escrevemos uma propriedade de CSS que o browser não reconhece, ele simplesmente ignora aquela linha e passa para a próxima. Isso acontece o tempo inteiro quando aplicamos as novidades do HTML. Lembra-se quando os browsers não reconheciam os novos tipos de campos de formulários do HTML5? O browser simplesmente substituía o campo desconhecido pelo campo comum de texto.

Isso é importante porque o que se faz hoje no desenvolvimento de um website, continuará funcionando de alguma forma daqui 10 anos. Como os browsers têm essa tolerância a falhas, linguagens como HTML e CSS ganham poder para evoluir o tempo inteiro, sem os bloqueios das limitações do passado.

Entender a importância do *Fault Tolerance* é a chave para entender o *Progressive Enhancement*. Na verdade, o *Progressive Enhancement* não seria possível se essa tolerância de falhas não existisse em browsers e outros meios de acesso.

Tudo sobre acessibilidade

Fundamentalmente, *Progressive Enhancement* é tudo sobre acessibilidade. Na verdade, o termo acessibilidade é normalmente usado para indicar que o conteúdo deve ser acessível para pessoas com necessidades especiais. O Progressive Enhancement trata isso mas na ótica de que todo mundo tem necessidades especiais e, por isso, o acesso ao conteúdo deveria ser facilitado para qualquer pessoa em qualquer tipo de contexto. Isso inclui facilmente pessoas que acessam websites via smartphones, por exemplo, cuja tela é pequena e algumas das facilidades que existem no desktops estão ausentes.

Níveis de Tolerância

Nós passamos por alguns níveis ao desenvolver algo tendo como método o *Progressive Enhacement*. Esses níveis têm como objetivo sempre servir, primeiro, o conteúdo e, depois, todas as funcionalidades e comportamentos que possam melhorar o consumo deste conteúdo e também de toda a página.

O objetivo por trás da tolerância de erros é a de sempre manter um fallback quando algo ruim acontecer. A primeira camada geralmente é um simples feedback básico, que será compatível com a maioria dos dispositivos. Esse feedback geralmente é servir um conteúdo em forma de texto. Isso é óbvio pois o texto é um conteúdo acessível para praticamente qualquer meio de acesso existente hoje. Muitos dos elementos do HTML têm um fallback de texto para casos em que elemento não seja carregado ou não seja reconhecido. Lembra do atributo `alt`? Até mesmo nas tags de vídeo e audio, como a seguir:

```
1 <video src="video.ogg" controls>
2   Texto de fallback.
3 </video>
```

A segunda camada é a semântica do HTML. Cada elemento do HTML tem sua função e, principalmente, seu significado. Eles acrescentam significados a qualquer informação exibida pelo HTML e, muitas vezes, estendem o significado que o texto sozinho não conseguiria.

A terceira camada de experiência é a camada visual, onde vêm o CSS e também as imagens, audios e vídeos. É onde a coisa fica bonita e interativa. Aqui, você sente mais a tolerância dos browsers a falhas. Usamos o tempo inteiro propriedades que nos ajudarão a melhorar a implementação de layouts, mas que, em browsers antigos, podem não ser renderizados. Experimentamos isso a todo momento.

A quarta camada é a camada de interatividade ou comportamento. O JavaScript toma conta dessa parte controlando os elementos do HTML, muitas vezes controlando propriedades do CSS para realizar ações de acordo com as interações do usuário.

A camada final é uma extensão da semântica dos elementos do HTML. Aí é onde inserimos as iniciativas de WAI-ARIA. É onde vamos guiar leitores de telas e outros meios de acesso para elementos e pontos importantes na estrutura que o layout se baseia. Indicando quais regiões e elementos são referência de navegação. Para saber mais sobre WAI-ARIA, veja o capítulo 6.

Nós podemos resumir as camadas em 3:

- Camada de conteúdo: HTML semântico e rico com WAI-ARIA e tags corretas;
- Camada de formatação: CSS e estilos;
- Camada de comportamento: JavaScript.

Estas são as três camadas clássicas sobre as quais você já ouviu durante sua vida profissional inteira. Não há segredo. Não há nada novo.

Costumo dizer para meus alunos que sempre que algo no desenvolvimento web — principalmente quando se trata com HTML/CSS — está ficando complicado, é porque há algo está errado. O HTML foi feito simples e vai continuar simples. Não há motivo para complicarmos algo que já é simples (estou olhando para você, dev back-end, que insiste em usar haml).

Fault Tolerance deve ser levado em conta em todos os seus projetos Web. Pensar assim dá flexibilidade para o projeto avançar para qualquer caminho, sem medo de problemas, pois esses problemas podem ser gerenciados e principalmente solucionados de forma eficaz.

Fazer websites compatíveis com todos os dispositivos do universo, com todos os sistemas de busca, leitores de tela e qualquer outra coisa que um dia utilizaremos é tudo sobre voltar às raízes do desenvolvimento Web. É ser simples desde o início. É tratar de poucas coisas por vez.

3.4 Esqueça os nomes Responsive e Adaptive

Agora esqueça tudo o que falei aqui e foque em apenas uma coisa: você não faz websites para dispositivos, você faz websites para os usuários. Seu objetivo é que os usuários consigam acessar a informação sem nenhuma restrição técnica. Isso quer dizer que ele pode querer acessar seu site de um relógio ou de uma TV, não importa, ele vai acessar, de preferência sem tantas mudanças estruturais para evitar novos aprendizados.

Sinceramente não importa qual tecnologia ou metodologia utilizada pela equipe. O que importa é um usuário falando bem do seu produto pois ele conseguiu encontrar o que queria. Por que ele chegou, fez o checkout sem problemas e não sentiu falta de informações entre as versões. O importante é a ubiquidade da informação e a satisfação final do usuário.

Isso é Web.

Sobre o autor

Diego Eis fundou o Tableless.com.br, website brasileiro que produz conteúdo sobre boas práticas e novas tecnologias de desenvolvimento web front-end. Treinou equipes de empresas como Whirpool, Oi, Accenture, Itaú, Nokia e outros. Agora está partindo para sua segunda empresa.

Capítulo 4

Tudo que você precisa saber para criar um framework de sucesso

"Não é possível obter uma criação coletiva a partir da soma de criações individuais. Uma criação é sempre resultado de um processo conjunto de criação."
– Horácio Soares Neto

Todo projeto tem uma origem, uma necessidade e um problema para resolver. Quando criamos um produto/serviço, queremos resolver um problema para alguém, ou somos ousados a ponto de criar um novo problema para alguém. Como desenvolvedores, nós podemos criar uma *checklist* de problemas predefinidos que temos ao iniciar um projeto, afinal todo projeto que se preze deveria ser preparado para ser acessível, semântico, performático, testável e sustentável. Há alguns anos, um grande problema começou a ganhar um pouco mais de popularidade: a reutilização de trechos de código. Seja HTML, CSS ou JavaScript, esses trechos, conhecidos como módulos, funcionam como peças de *LEGO* que, somadas estrategicamente, criam projetos flexíveis, padronizados e organizados.

Por que é tão difícil criar um padrão?

O ser humano tem uma enorme facilidade no reconhecimento de padrões, pois é o que usamos para viver e é assim que funcionamos.

Nosso cérebro é formado de duas áreas responsáveis pelo armazenamento da memória. A primeira delas é responsável pela memória de curta duração ou memória de trabalho; e a segunda é a memória de longa duração. Como a memória curta possui uma capacidade limitada, existe uma camada anterior conhecida como **armazenamento sensorial**, responsável pela triagem do que será armazenado ou não. Para isso, ela precisa definir em segundos o que vai armazenar na memória de curta duração e o que ela vai esquecer. O que não for armazenado pelo sistema de gerenciamento de memória é ignorado, gerando um reaprendizado o tempo todo sobre aquilo que não foi guardado.

É como se essa primeira camada da entrada de informação fosse um cache do browser, pois a cada vez que você fechar o browser ou reiniciar o seu computador, ela é automaticamente limpa. Claro que se a sua ida ao mesmo site for constante, você automaticamente terá o site na página inicial do seu browser, em uma espécie de memória de curta duração, ou o colocará em seus favoritos, uma espécie de memória de longa duração.

A maior dificuldade para os manipuladores da mente, como designers, é conseguir tornar o reconhecimento das ações, objetivos e interações na interface tão simples e intuitivos a ponto de o usuário relacionar tal comportamento com algo que está guardado na memória de longa duração, também conhecida como memória semântica. Por exemplo, o slide de uma foto com um dedo deslizando, um logo no canto superior esquerdo que o leva para a página inicial, ou reconhecer que o ícone de um lápis significa editar. Quanto mais intuitivo, menos a memória de trabalho precisa ficar trabalhando, dando uma sensação de mais segurança, menos esforço e mais conforto para o usuário.

Sabendo, então, que o reconhecimento dos padrões é feito baseado na experiência de armazenamento de cada ser humano, como uma pessoa pode se achar suficiente para criar os padrões que serão utilizados por toda uma equipe, local, remota e às vezes até internacional? Criar padrões é algo extremamente difícil.

Criação coletiva, um projeto de muitos

Lembro de quando trabalhava na Petrobras, em que toda semana tínhamos uma reunião de aproximadamente 3 horas para rever e criar os padrões da equipe. Nossa

meta era documentar todos os componentes utilizados, de forma que conseguíssemos mapeá-los em suas variações e dar início a um planejamento de corte por toda a equipe, o que geralmente acontecia em um quadro branco. No fim, um dos desenvolvedores da equipe era responsável por criar esse componente seguindo o padrão criado e documentado.

O projeto, carinhosamente chamado de "zord", era um grande framework que unia todos os componentes utilizados nos sistemas internos da nossa gerência (mais de 20 sistemas). Nele, tínhamos componentes predefinidos como Abas, Sanfonas, Navegações, Tabelas, Botões, Grids e Formulários. O projeto já passou por muitas atualizações e está sendo utilizado pela equipe até hoje nos projetos da Petrobras.

Desde 2003, no começo da minha carreira dentro de empresas, a ideia de padronizar componentes sempre esteve presente em mim. Sempre fui um metódico compulsivo, que guardava referências em pastas separadas por componentes, e não eram só referências: eu criava códigos desses artefatos separados para que todo o time tivesse acesso e perpetuasse esse grande repositório. No caso da Petrobras, quando comecei a fazer parte do time de produto, identifiquei que existia um grande retrabalho para cada projeto criado. Acredito que qualquer retrabalho é um sintoma, pois dá visibilidade a um problema de processo, seja causado por gargalos, insegurança, ego, preguiça ou falta de experiência. Assim sendo, comecei a mapear todos os componentes que eram utilizados em mais de 2 projetos, para começar a intermediar perante a equipe uma forma de padronizarmos o artefato.

Vale lembrar que a padronização também envolve o processo de criação, então a reunião envolvia padrões de design que também não eram contemplados pelo time, atuando na época basicamente como uma fábrica de software.

Depois de algumas semanas estudando o processo, eu, Daniel Araujo e Marcello Manso começamos a criar um framework interno que prometia revolucionar a forma como criávamos formulários para os sistemas internos da Petrobras. O ponto alto do projeto foi o Grid criado, pois, para ajustarmos nosso formulário para qualquer container de qualquer sistema, teríamos que criar uma solução adaptativa em porcentagem. Isso me levou a uma imersão dentro do grid 960 criado por **Nathan Smith**, até então o GRID mais importante do mundo quanto à diagramação Web. Adaptando o sistema de grid criado por Nathan, criamos um framework fluido baseado em 12 colunas, proporcionando-nos muita agilidade e flexibilidade na criação de formulários Web. Acredito que isso só foi possível por causa da criação da propriedade `box-sizing`, que permitiu fazer um input com `padding` e `border` ocupar 100% da área de um container.

O framework deixou um legado tão grande para a Petrobras que ganhamos 3 prêmios de criação no ano. Tal reconhecimento nos fez perceber que outras pessoas e empresas podiam e estariam passando pelos mesmos problemas. Logo, criamos o *Formee*, um framework que foi muito famoso em 2009/2011 e bateu a meta de mais de 70 mil downloads em 2012. Infelizmente, com o boom de novos frameworks, o Formee ficou um tanto quanto desatualizado, mas quem sabe não volte em breve com força total, não é verdade?

4.1 Guerra contra os frameworks prontos

Comumente sou perguntado sobre ser contra o Twitter Bootstrap e Zurb Foundation. Como eu sempre falo, um framework pode ser o responsável pelo sucesso do seu site e da sua aplicação e lhe poupar milhares de dólares. Porém, mal utilizado, pode representar o gargalo do desenvolvimento do seu produto e levar o seu projeto ao fracasso.

Muitas vezes, as pessoas associam a escolha da utilização de um framework ao ganho de tempo. Eu entendo que tempo é a coisa mais crítica dentro de um projeto, principalmente startups, porém, eu sempre gosto de utilizar o *"reductio ad absurdum"* para lembrar que fatiar um layout utilizando o Photoshop ou Fireworks também é otimização de tempo, assim como utilizar sites prontos.

Na maioria dos casos, o desenvolvedor não inspeciona o código do framework, passando a criar um outro arquivo CSS para customizar o seu projeto. Em muitos outros casos, o design do projeto é influenciado pelo framework, passando a ter limitações quanto à escolha de espaçamentos, grid e outras variações que já estão predefinidas no framework. Tudo isso é uma inversão de valores quanto a um projeto, pois agora que um time de produto está mais integrado do que nunca, trabalhando com metodologias ágeis, documentações e padrões definidos em equipe, como podemos aceitar que um código de um projeto – que é o *core* da sua aplicação – está sendo definido por um framework terceiro só por agilidade?

Por conta desses pensamentos, **defendo a criação do seu próprio framework**, em que o time analisará os artefatos presentes na aplicação, padrões de espaçamentos, grids, fontes, palheta de cores e interações serão os guias para a criação de módulos e um projeto reutilizável e sustentável. Mas para isso você precisará definir se você quer realmente controle total sobre o seu código. Você não consegue criar um código mais simples, com menos objetos e dependências do que os frameworks prontos? Você não acredita que, em pouco tempo, um framework próprio, focado

no problema que o site precisa resolver, será mais eficaz? Se as respostas forem sim, você está a um passo de aprender a criar o seu próprio framework.

4.2 Organização e padronização

Pensar em aplicações é muito comum para programadores back-end e até mesmo programadores JavaScript, mas tenho observado um grupo muito grande de desenvolvedores front-end que atuam mais nessa camada estrutural de HTML e CSS, por isso não podemos e nem vamos ignorá-los. Vamos começar pensando sobre a organização do projeto.

Para discutirmos sobre organização, é necessário rever seus atuais padrões em projeto. Responda essas perguntas sobre o seu projeto:

1) Qual o padrão de comentários por seção e inline?

2) As classes do seu projeto são feitas com hífen (-), underscore (_), ou camelCase (aA)?

3) Que tipo de formato de cores você usa no seu projeto? Hexadecimal? HSL? RGB? RGBA? Nome da cor?

4) Comentários, classes e documentações estão em que idioma? Português? Inglês? Misto?

Para componentizar sua página de maneira sustentável, você precisa explorar ao máximo a organização e padronização do projeto. O problema na verdade está em nosso histórico como desenvolvedores, pois sempre fomos acostumados a desenvolver da seguinte forma:

- reset (zerar os estilos para iniciar um novo projeto);
- estrutura (criar a estrutura base do projeto, geralmente composto por header, main, footer e colunas macro);
- página (definir customizações para cada página do projeto).

Esse modelo tem sua utilidade, mas desenvolver através de fluxo de páginas é muito cruel com o CSS, pois passamos sempre a incrementar o código, sem observar o que já foi criado e o que pode ser reutilizado. Assim, um botão feito na página 1

necessita de alterações para a página 2, e por conta disso não poderá ser reutilizado na página 3. Depois de um tempo, ninguém entenderá o seu CSS, nem você mesmo, passando a viver triste e sozinho, abandonado no mundo das folhas de estilo sem organização.

Para quem está começando, recomendo sempre utilizar o modelo mais básico de um projeto organizado:

- reset (zerar os estilos para iniciar um novo projeto);
- base (criar a base do projeto, geralmente composto por header, main, footer e demais estruturas base);
- modules (define o estilo para cada componente de forma isolada, assim como seus estados de interação);
- layout (faz as mudanças que um componente ou que a base pode ter para se ajustar a uma página ou contexto específico).

Nesse caso, teoricamente acabamos com o conceito de desenvolvimento de páginas e passamos a tratar qualquer projeto como uma aplicação. Infelizmente, ainda é necessário utilizar um arquivo como o "layout" para funcionar como um coringa, a fim de permitir customizações por questões de layout de página. Mas, quanto menos utilizá-lo, mais modularizado e sustentável será o seu projeto.

Existe ainda uma outra forma de criar o seu modelo de projeto, focado em tematização:

- reset (zerar os estilos para iniciar um novo projeto);
- base (criar a base do projeto, geralmente composto por header, main, footer e demais estruturas base);
- theme (estilizar o projeto quanto ao visual temático, como cores, fundos e fontes).

Essa estratégia temática já foi utilizada em diversos projetos, inclusive por mim mesmo, tanto na criação inicial do Formee quanto na primeira versão do Conf boilerplate (https://github.com/braziljs/conf-boilerplate) . Porém, eu ainda sinto que colocar todos os componentes na "base" é bastante confuso e contraproducente. Sendo assim, passei a adotar um modelo mais completo para organização de projeto:

- reset (zerar os estilos para iniciar um novo projeto);
- base (criar a base do projeto, geralmente composto por header, main, footer e demais estruturas base);
- components (define o estilo para cada componente de forma isolada, assim como seus estados de interação);
- form (define o estilo pra todos os objetos de formulário, assim como seus estados de interação);
- typography (define o estilo para todos os os objetos textuais, assim como fontes externas e *icon fonts*).

O interessante desse modelo é a atenção que você pode dar a um componente específico. Por exemplo, caso um projeto tenha uma gama de variações muito grande de um componente botão, você pode dar independência a ele, isolando-o do item "components". Alguns exemplos de componentes que, dependendo de suas complexidades e tamanhos, podem virar um projeto próprio:

- buttons
- table
- tabs
- news
- post
- grid

4.3 Nomenclatura

Uma das coisas em que eu mais presto atenção ao olhar um código é a nomenclatura utilizada para definir as classes de um projeto. Veja bem, digo apenas classes, pois não vejo sentido em utilizar ID para estilização, visto que uma classe pode ser altamente reutilizada e o ID só pode haver um por HTML. Sabendo que cada um pode ter o seu próprio estilo de nomear objetos, mostrarei o meu padrão de nomenclatura para um botão.

```
<!-- botão -->
<a class="btn"></a>
```

Perceba que o botão recebe o nome de `btn`, um prefixo que funciona bem para português ou inglês. Sempre que possível, eu utilizo três letras no prefixo, seguido de hífen para separar as palavras do nome. Veja a seguir:

```
<!-- botão primário -->
<a class="btn btn-primary"></a>
```

Esse é o ponto em que algumas pessoas divergem. Muitos dizem que, a partir do momento em que utilizamos CSS3, podemos utilizar os novos seletores para resgatar o prefixo "btn-", não precisando repetir a escrita no HTML. Caso você não conheça, eis os dois seletores CSS3 citados como soluções:

```
/* todos os itens que COMECEM na classe com o prefixo "btn-" */
[class^=btn-] {}

/* todos os itens que POSSUEM na classe o prefixo "btn-" */
[class*=btn-] {}
```

Eu creio que essa solução seja muito custosa para o CSS, não só de escrita, aceitação, como também de performance. Assim, prefiro orientar as classes ao objeto que eu quero estilizar, garantindo melhor performance, intuitividade e documentação dos componentes. Sobre performance, segue uma lista criada por **Steve Souders**, que mostra como os *regex selectors* baseados em atributos são ruins para performance no browser:

1) ID, e.g. `#header`

2) Class, e.g. `.promo`

3) Type, e.g. `div`

4) Adjacent sibling, e.g. `h2 + p`

5) Child, e.g. `li > ul`

6) Descendant, e.g. `ul a`

7) Universal, i.e. `*`

8) Attribute, e.g. `[type="text"]`

9) Pseudo-classes/-elements, e.g. `a:hover`

> Embora o ID seja o seletor mais performático, o fato de não ser reutilizável (explicado anteriormente) é o bastante para adotar classes como seletores principais do seu projeto.

Veja mais algumas variações de classes para o elemento botão.

`btn-primary, btn-secondary, btn-delete, btn-edit, btn-disabled`

O ponto-chave na hora de criar uma classe é focar sempre em sua função. Porém, existem pequenas exceções que são adotadas pelo mercado, por exemplo tamanhos predefinidos de botão como `btn-small`, `btn-large`, `btn-full`.

> Na maioria dos projetos, algumas classes se tornam tão genéricas e reutilizáveis, que deixam de necessitar de prefixo, como `clear`, `omega`, `alpha`, `small`, `large`, `full` etc.

4.4 Regras de estado

Cada vez mais aplicações têm tido mais interações com o usuário no client. Essas interações criam uma experiência cada vez mais rica e fantástica para o usuário. O problema é que muitas dessas interações são feitas com uma manipulação *inline* do elemento, como um JavaScript desabilitar um botão, atribuindo CSS inline para o elemento. Ou ainda um botão que possui um ícone dentro, só que sem nenhuma marcação que me permita controlar pelo CSS todos os outros botões que também possuam ícones.

O controle dos estados dos elementos é conhecido como **state rules** e pode ajudar bastante a documentar e controlar um projeto.

1) O usuário está vendo uma galeria de imagens. Ao clicar no botão "ver mais fotos", o site carregará via ajax mais itens, deixando um *loading* no botão até o final da operação. Qual a classe recomendada para um botão no momento do *loading*?

Tudo indica que a escolha mais óbvia seria `btn-loading`. O problema é que o prefixo indica que é um tipo de botão e não um estado do mesmo, assim sendo, recomenda-se a utilização de dois dos mais conhecidos *state rules*, o `is-` e `has-`.

O `is-` indica o estado do elemento, geralmente para um estado provisório, ou que pode ser alterado a qualquer momento pelo usuário:

`is-loading`, `is-disabled`, `is-active`, `is-checked` etc.

O `has-` indica o estado de pertencimento do elemento, geralmente para controlar diagramações ou espaçamentos:

`has-photo`, `has-icon`, `has-comments`, `has-reviews`, `has-no-photo`, `has-no-comments` etc.

Vamos considerar um sistema de notícias, no qual temos variações da notícia com/sem imagem e com/sem botão. Nossa estrutura principal é assim:

```
<div class="news-item">
    <div class="news-content">
        <h1>Biscoito Globo</h1>
        <p>Muitas vezes amigos de outras cidades me perguntam qual o
        principal prato do Rio de Janeiro, eu sempre respondi feijoada,
        mas creio estar enganado. A principal iguaria carioca é o famoso
        biscoito Globo. Embora o biscoito de polvilho tenha sido
        inventado em Minas Gerais e os irmãos criadores do biscoito
        terem criado o mesmo em São Paulo. Então, se tem algo que você
        não pode deixar de experimentar ao visitar a cidade maravilhosa
        é tanto o biscoito salgado quanto o biscoito doce Globo,
        acompanhado de um mate de galão, você não vai querer voltar para
        sua cidade.</p>
    </div>
</div>
```

Ao incluir uma imagem, nosso código viraria:

```
<div class="news-item">
    <div class="news-media">
        <img src="biscoito-globo.jpg" alt="Biscoito Globo">
    </div>
    <div class="news-content">
        <h1>Biscoito Globo</h1>
        <p>Muitas vezes amigos de outras cidades me perguntam qual o
        principal prato do Rio de Janeiro, eu sempre respondi feijoada,
        mas creio estar enganado. A principal iguaria carioca é o famoso
```

```
        biscoito Globo. Embora o biscoito de polvilho tenha sido
        inventado em Minas Gerais e os irmãos criadores do biscoito
        terem criado o mesmo em São Paulo. Então, se tem algo que você
        não pode deixar de experimentar ao visitar a cidade maravilhosa
        é tanto o biscoito salgado quanto o biscoito doce Globo,
        acompanhado de um mate de galão, você não vai querer voltar para
        sua cidade.</p>
    </div>
</div>
```

O problema nesse exemplo é que o item `news-media` passou a ocupar parte da diagramação, assim o `news-content` vai deixar de ocupar 100% de largura para dividir espaço com o `news-media`. Eu posso ter vários `news-item` em uma lista. Como estamos reutilizando as classes, o que eu precisaria fazer para manter um `news-item` sem imagem e outro com imagem? Simples! Utilizando as *Regras de estado*, veja a seguir:

```
<div class="news-item has-img">
    <div class="news-media">
        <img src="biscoito-globo.jpg" alt="Biscoito Globo">
    </div>
    <div class="news-content">
        <h1>Biscoito Globo</h1>
        <p>Muitas vezes amigos de outras cidades me perguntam qual o
        principal prato do Rio de Janeiro, eu sempre respondi feijoada,
        mas creio estar enganado. A principal iguaria carioca é o famoso
        biscoito Globo. Embora o biscoito de polvilho tenha sido
        inventado em Minas Gerais e os irmãos criadores do biscoito
        terem criado o mesmo em São Paulo. Então, se tem algo que você
        não pode deixar de experimentar ao visitar a cidade maravilhosa
        é tanto o biscoito salgado quanto o biscoito doce Globo,
        acompanhado de um mate de galão, você não vai querer voltar para
        sua cidade.</p>
    </div>
</div>
```

Assim, eu poderia fazer a seguinte configuração:

```
.news-media {
    width: 28%;
    margin-right: 2%;
```

```
}
.has-img .news-content {
    width: 100%;
}
```

Adicionando um botão para acessar o site, usaríamos o mesmo artifício:

```
<div class="news-item has-img has-btn">
    <div class="news-media">
        <img src="biscoito-globo.jpg" alt="Biscoito Globo">
    </div>
    <div class="news-content">
        <h1>Biscoito Globo</h1>
        <p>Muitas vezes amigos de outras cidades me perguntam qual o
        principal prato do Rio de Janeiro, eu sempre respondi feijoada,
        mas creio estar enganado. A principal iguaria carioca é o famoso
        biscoito Globo. Embora o biscoito de polvilho tenha sido
        inventado em Minas Gerais e os irmãos criadores do biscoito
        terem criado o mesmo em São Paulo. Então, se tem algo que você
        não pode deixar de experimentar ao visitar a cidade maravilhosa
        é tanto o biscoito salgado quanto o biscoito doce Globo,
        acompanhado de um mate de galão, você não vai querer voltar para
        sua cidade.</p>
    </div>
    <div class="news-btn">
      <a class="btn" href="http://www.biscoitoglobo.com.br">Ver site</a>
    </div>
</div>
```

Adicionando a possibilidade do `has-btn` ao projeto, aumentaríamos a complexidade da configuração:

```
.news-media {
    width: 28%;
    margin-right: 2%;
}
.news-btn {
    width: 28%;
    margin-left: 2%;
}
.has-img .news-content,
.has-btn .news-content {
```

```
    width: 70%;
}
.has-img.has-btn .news-content {
    width: 40%;
}
```

Perceba que a notícia está preparada para receber apenas o `has-img`, mudando o content para 70%. Caso ela receba apenas o `has-btn`, o content também mudará para 70%. Porém, se receber os dois, `has-img` e `has-btn` ao mesmo tempo, ele configura o content para 40%. Incrível e fácil né?

Trabalhar com prefixos focados na função e pensar nos estados dos elementos nos faz diminuir bastante o risco de um projeto, além de garantir mais qualidade para criarmos os próximos artefatos.

4.5 Modularização: Pensando em seu projeto como um sanduíche

"Ao separar um sanduíche em pedaços menores, você pode rapidamente e eficientemente criar uma infinidade de variedades. Assim que as classes deveriam ser baseadas. Em vez de "#sanduiche" agora temos ".pao.alface.queijo.salame.tomate.maionese" e isso é incrível! Não gosta de tomate? É fácil deixá-lo fora ... Classes baseadas em estruturas é como comer no Subway (só que melhor).

– Harry Roberts

Lembro de uma época em que designers projetavam páginas e, depois de semanas, elas eram entregues para o desenvolver front-end "cortá-las". Se você ainda segue esse método de trabalho, está mais do que na hora de se atualizar e se adequar ao mercado ágil. Hoje em dia, criamos *features*, componentes padronizados que são ajustados em interfaces, de acordo com o fluxo do usuário.

O designer passa a projetar fluxos e seus artefatos, passando de forma iterativa para o desenvolvedor que sempre "corta" os componentes isoladamente, podendo ser utilizados e reutilizados em qualquer lugar da aplicação, não importa qual a página ou seção.

Esse novo modelo de "corte" é o que deu origem ao polêmico *"CSS Orientado ao objeto"*. Como vimos anteriormente, os prefixos são vinculados exclusivamente à função de cada objeto, acompanhados de suas extensões. Vimos a aplicação do botão, vejamos alguns outros objetos e suas possíveis classes:

tooltip: tooltip, tooltip-pin-down, tooltip-pin-up, tooltip-small, tooltip-header, tooltip-content.

tabs: tabs-nav, tabs-item, tabs-link, tabs-content.

Vamos criar um artefato de navegação com as classes a seguir:

nav: nav-list, nav-item, nav-link.

```
<nav class="nav">
    <ul class="nav-list">
        <li class="nav-item">
            <a href="#" class="nav-link"></a>
        </li>
        <li class="nav-item">
            <a href="#" class="nav-link"></a>
        </li>
        <li class="nav-item">
            <a href="#" class="nav-link"></a>
        </li>
    </ul>
</nav>
```

1) Precisamos desenvolver um componente de mensagens, como você faria?

> Eu cometia um erro em meus códigos ao utilizar o componente de mensagem vinculado ao componente de formulário. Felizmente, fui percebendo que mensagem é um componente isolado, pois aparece ao longo do sistema e em diferentes momentos. Sendo assim, passei a utilizar o nome alert para o elemento de mensagem.
>
> Uma outra mudança que fiz nesse componente foi a retirada dos prefixos das customizações. Por exemplo, antigamente utilizava msg-error para o componente msg, mas percebi que palavras como *success*, *error*, *information* e *warning*, são bastante reutilizáveis em outros componentes. Assim sendo, passei a usar apenas error na componente de mensagem – você pode utilizar "alerta", se preferir.

Tipografia

"Text is a UI"
– Jakob Nielsen

Muitas vezes esquecemos da importância da tipografia em uma interface, só que ela, possivelmente, é o componente mais crucial do seu projeto. Precisamos definir os elementos textuais da aplicação. Eles geralmente são títulos, subtítulos, listas, parágrafos e links. É bastante importante você simular todas as combinações possíveis na etapa de testes, pois sempre pode ocorrer algum incômodo para o usuário. Veja o código a seguir:

```
<h2>Lorem ipsum dolor sit.</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Modi,
fugit voluptatem vero rem nesciunt corrupti accusamus fuga enim ut
autem. Molestias, ipsam dolores inventore officia quod cupiditate
repellendus labore ipsa.</p>
<h2>Lorem ipsum dolor sit.</h2>
```

Ele não é a mesma coisa que este:

```
<h2>Lorem ipsum dolor sit.</h2>
<h3>Lorem ipsum dolor sit.</h3>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Modi,
fugit voluptatem vero rem nesciunt corrupti accusamus fuga enim ut
autem. Molestias, ipsam dolores inventore officia quod cupiditate
repellendus labore ipsa.</p>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Modi,
fugit voluptatem vero rem nesciunt corrupti accusamus fuga enim ut
autem. Molestias, ipsam dolores inventore officia quod cupiditate
repellendus labore ipsa.</p>
<h3>Lorem ipsum dolor sit.</h3>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Modi,
fugit voluptatem vero rem nesciunt corrupti accusamus fuga enim ut
autem. Molestias, ipsam dolores inventore officia quod cupiditate
repellendus labore ipsa.</p>
```

Um subtítulo pode funcionar muito bem ao se relacionar com um parágrafo, mas pode ficar terrível ao se relacionar diretamente com outro subtítulo, ou ao se utilizar o mesmo para parágrafos únicos e parágrafos consecutivos. O ideal é você testar todas essas combinações a fim de identificar um meio termo de espaçamentos e margens.

> Acredito que toda tela de uma aplicação tenha o seu título da página específica. Esse título tende a ter um tratamento e espaçamento para o restante do conteúdo diferenciado. Para não impactar todos os outros subtítulos, eu crio uma classe `page-title` para esse elemento, fazendo as diferenciações para essa função destacada, deixando de influenciar as demais aplicações da mesma tag.

Testar a tipografia de um projeto é analisar suas variações e combinações com outros elementos. Para você conhecer um pouco mais sobre tipografia na parte de front-end, recomendo a wiki do **Normalize.css**, criado pelo Nicolas Gallagher (https://github.com/necolas/normalize.css/wiki) , que retrata a falta de padronização na mostragem dos elementos textuais entre os browsers. O Normalize é um framework bastante famoso, responsável pela reinicialização do CSS para todos os navegadores, preservação dos padrões de estilo e resolução de inconsistências entre os elementos tipográficos.

GRID

Criar um GRID é talvez uma das tarefas mais difíceis na hora de se criar um projeto, não é à toa que ele é responsável pelo fato de a maioria das pessoas utilizar frameworks prontos. Eu sempre lembro de quando criei o Formee e como foi difícil criar cálculos e mais cálculos para chegar ao Grid flexível perfeito, pois o Grid é uma parte perigosa do projeto, principalmente por exigir algumas escolhas desde o começo, como:

- Usarei `float` para diagramação?
- Usarei algum pré-processador como Less, Sass ou Stylus?
- Usarei colunas em pixels ou colunas flexíveis?
- Minhas colunas serão responsivas?

Em Fevereiro de 2014, fiz um projeto de experiência na empresa e o ganhador passou 3 dias comigo, ajudando-me a criar o novo framework de um produto. Quando começamos a definir o GRID, dei a tarefa de analisar 3 dos frameworks de GRID mais famosos do mercado para saber qual utilizaríamos. Veja as opções:

Twitter Bootstrap v2

```
<div class="row">
  <div class="span4">...</div>
  <div class="span3 offset2">...</div>
</div>
```

Bootstrap v3

```
<div class="row">
  <div class="col-md-8">...</div>
  <div class="col-md-4">...</div>
</div>
```

Zurb Foundation

```
<div class="row">
  <div class="small-6 large-2 columns">...</div>
  <div class="small-6 large-8 columns">...</div>
  <div class="small-12 large-2 columns">...</div>
</div>
```

Semantic gs

```
@column-width: 60;
@gutter-width: 20;
@columns: 12;

header { .column(12); }
article { .column(9); }
aside { .column(3); }
```

Bootstrap LESS variables

```
@grid-columns: 12;
@grid-gutter-width: 30px;
@grid-float-breakpoint: 768px;

header {.make-lg-column(12);}
article {.make-lg-column(9);}
aside {.make-lg-column(3);}
```

Não gosto do nome das classes do Bootstrap v2, nem da não separação entre o nome e o número. O Foundation e a nova versão do Bootstrap já possuem um

nome mais elegante e mais organizado. O que mais me incomoda nesses sistemas de GRID é a utilização de "row" para incluir as colunas. Você é obrigado a penalizar o seu HTML, incluindo tags sem valor algum, simplesmente para conseguir fazer linhas de colunas. Definitivamente, criar um framework de GRID é analisar bastante o custo x benefício de poluir algo em função de um sistema reutilizável. Dentro dessas opções, escolhemos o Semantic.gs (http://semantic.gs) que, diferente dos demais, não engessa quantidade de colunas na marcação, não polui o HTML e é extremamente configurável através de pré-processadores.

> Como vocês podem ver, o Bootstrap 3 agora também possui a possibilidade do sistema de GRID vinculado a variáveis em LESS, porém, pela elegância e por ser focado apenas em GRID, achamos o semantic mais interessante para o projeto.

A escolha do GRID tem que ser feita por projeto e, novamente, recomenda-se que envolva a equipe em testes para a tomada de decisão, até mesmo se a decisão for a criação do seu próprio sistemas de GRID, como fiz no Formee.

> Uma dica para quem está estudando o desenvolvimento de GRIDs é pesquisar sobre o atributo CSS `box-sizing` que é responsável por mudar o display do *box model*, passando a considerar o `padding` e `border` na hora de aparecer na largura/altura final. O *box-model* convencional não os considera na largura e altura, somando no resultado os valores, ou seja, 300px de largura acaba se tornando 300px + 2px de borda + 10px de padding = 312px total.

4.6 Agrupando seus componentes em um único local

A melhor maneira de desenvolver os componentes isolados de um projeto é não desenvolvê-lo na página em questão e, sim, em um local separado. Esse local reúne todos os componentes reutilizáveis do projeto, funcionando como um espaço de documentação, guia e repositório de componentes. Veja por exemplo, o **"UI kit"** do Twitter Bootstrap (http://getbootstrap.com/components/) e do Zurb Foundation (http://foundation.zurb.com/docs/) .

Você precisa criar essa página de apresentação onde listará os componentes padronizados. Em um primeiro momento, você pode inserir os componentes na página, agrupando da maneira que achar mais organizada (veja os links anteriores para inspirar-se). Essa etapa lhe dará mais segurança, pois a cada componente finalizado, você e sua equipe comemorarão o padrão criado. Lembre-se que, mais do que criar os componentes, é preciso sempre voltar para melhorar seus artefatos, identificando pontos de melhoria ou bugs, tratando e atualizando-os nessa mesma página de padrões.

Element.css

Foi lançado um projeto chamado Element CSS (http://elementcss.com) focado em ajudar e otimizar a criação de um guia em cima dos componentes do seu projeto. O projeto organiza seus componentes online, assim seu time pode se logar e visualizar os padrões em qualquer lugar conectado a internet, além de criar novos componentes ou variações sempre que quiser. Um outro recurso interessante é poder baixar a qualquer momento o CSS compilado de todos os componentes. Para projetos grandes, creio ser um pouco limitado, mas para quem está começando, pode ser uma bela mão na roda.

O designer moderno já projeta o site ou aplicação através de um *"UI kit"*, ficando mais fácil para o desenvolvedor recriá-lo e documentá-lo. Para que alguns componentes se encaixem dentro da diagramação desse guia, recomendo criar um CSS interno dentro desse documento com variações apenas para compor melhor o kit (puramente estético). Costumo separar cada componente dentro de seções, então veja um exemplo de um CSS dentro do *"UI kit"*:

```
.ui-kit {
    padding: 50px;
}

/* SECTIONS */
.section {
    position: absolute;
    top: 50px;
    margin-right: 50px;
}
```

```
.section-header {
    position: absolute;
    top: 50px;
    left: 50px;
}
.section-btn {left: 330px;}
.section-texts {
    left: 330px;
    top: 200px;
}
.section-progress {
    left: 330px;
    top: 550px;
    width: 400px;
}
.section-form {
    left: 630px;
    top: 700px;
    right: 0;
}
.section-alerts {
    left: 50px;
    top: 700px;
}
```

Juntar seus principais componentes em um único arquivo mudará a sua forma de pensar e escrever HTML e CSS. Os principais benefícios de se criar um documento separado são:

- Ver se seu o componente funciona sem herança CSS;
- Ter seus módulos documentados;
- Propor reutilizações, ao invés de desenvolver páginas;
- Propor análises 360 nos componentes, sugerindo melhorias isoladas;
- Atingir o DRY CSS (*don't repeat yourself*).

4.7 Tornando-se o rei dos padrões

O benefício da modularização é incrível e já tive a felicidade de ver ganhos em diversos projetos em que eu atuei. Alguns componentes meus em muitos projetos são realmente idênticos. Assim sendo, eu posso copiar tanto o HTML quanto o CSS, fazendo pequenas alterações. Mais do que isso, eu posso até mesmo criar *snippets* em meu editor, para, com um simples atalho, gerar o componente desejado. Isso só é possível fazer com um amadurecimento de padrões e escrita.

Certa vez, estava dando um treinamento para uma instituição de ensino e um dos exercícios práticos era dividir a turma em 2 grupos, sendo que eles teriam 5 minutos para criar o planejamento de corte da página do Facebook, determinando os componentes e criando o padrão de classes. O resultado foi surpreendente! O primeiro grupo criou as classes focadas nos componentes, utilizando `btn-` e `comment-`, enquanto o outro grupo criou classes focando nas ações `edit-` e `link-`. No fim, peguei o capitão de cada grupo e coloquei para explicar o planejamento produzido pelo outro time. Reconheço que foi engraçado ver as tentativas de ler e entender padrões como `b-` e `f-`, ainda mais sabendo que alguém de fora nunca pode julgar que o padrão de um time não está certo.

Não existe padrão errado, assim como não existem 2 padrões. Existe a máxima: se você tem mais de um padrão, ele não é um padrão. Então, para se tornar o rei dos padrões, você precisa pensar que você não está criando um projeto para você mesmo e, sim, para o outro.

Em 2009, eu já bradava em algumas palestras que você não tem que criar um código que só você entende, achando que isso o fará necessário para a empresa ou projeto que está atuando. Caso um próximo desenvolvedor assuma o projeto ou o seu cargo, ele e a empresa têm que se lembrar de você pela sua qualidade de escrita e de organização, e não pelo seu egoísmo de um código não preparado. Agora que você já iniciou seu treinamento para se tornar o rei dos padrões, vou deixá-lo praticar com seu editor e qualquer dúvida é só me procurar. Boa sorte!

Sobre o autor

Bernard De Luna atua em projetos digitais desde 1999, já participou de diversos projetos do Brasil, EUA, Inglaterra, França e Austrália. Nos últimos anos liderou time de produtos na Petrobras, foi diretor criativo da Melt DSP, coordenador de produto da Estante Virtual e atualmente é Head de produto no Videolog e co-founder do Freteiros.com. Especializado em Front-end, Design funcional e Marketing Digital, Bernard já deu mentoria na Lean Startup Machine, é mentor da Papaya Ventures e da Codemy, já palestrou em mais de 70 eventos pelo Brasil, além do evento internacional do W3C, Startup Camp LATAM e HTML5DevConf, entre outros. No tempo vago, adora tirar foto do seu cachorro com a hashtag #jupidog, tem uma mania feia de falar em terceira pessoa e se autointitula "sexy web projects specialist".

Capítulo 5

Tornando a web mais dinâmica com AngularJS

5.1 Por que AngularJS

O JavaScript evolui de forma significativa com a criação de novos frameworks, tornando-se, a cada dia, mais poderoso e utilizado entre os desenvolvedores. Um deles é o jQuery, amplamente conhecido e praticamente obrigatório se você deseja controlar os elementos de uma página HTML, também chamado de DOM.

Seguindo essa linha, diversos frameworks de qualidade continuam surgindo, e dentre eles temos o AngularJS, que será o nosso principal objeto de estudo.

Conhecendo AngularJS

AngularJS é construído sobre a ideologia de que a programação declarativa deve ser usada para construção de interfaces e componentes, enquanto que a programação imperativa é excelente para escrever as regras de negócio.

O framework se adapta e estende o HTML tradicional para uma melhor experiência com conteúdo dinâmico, com a ligação direta e bidirecional dos dados (os famosos *data-binding*). Isso permite sincronização automática de models e views, abstraindo a manipulação do DOM e melhorando os testes.

AngularJS conseguiu atrair muita atenção nesses últimos tempos, principalmente devido ao seu sistema inovador de modelagem, facilidade de desenvolvimento e práticas de engenharia muito sólidas. Alguns dos objetivos do AngularJS são:

1) Abstrair a manipulação do DOM da lógica do app. Isto melhora os testes;

2) Considera os testes do app tão importantes quanto seu desenvolvimento. A dificuldade do teste é diretamente afetada pela maneira como o código é estruturado;

3) Abstrai o acoplamento entre o lado cliente e o lado servidor da aplicação. Permitindo que o desenvolvimento do app evolua em ambos os lados, de forma paralela, e permite o reúso de código.

A versão 1.0 foi lançada apenas em junho de 2012. Na realidade, o trabalho começou em 2009 como um projeto pessoal de Miško Hevery, um funcionário do Google. A ideia inicial acabou sendo tão boa que, no momento da escrita, o projeto foi oficialmente apoiado pelo Google, e há toda uma equipe do Google trabalhando em tempo integral no AngularJS. O projeto é open source e está hospedado no GitHub (https://github.com/angular/angular.js) . É licenciado pela Google sobre os termos de licença MIT.

5.2 AngularJS

Para usar a estrutura AngularJS são necessárias duas alterações em nosso documento HTML, PHP, Rails etc.

A primeira coisa a se fazer é incluir a biblioteca em nosso documento. A segunda, é incluir a propriedade `ng-app` no elemento `html` em que queremos "ativar" o aAgularJS. Neste caso, começamos inserindo na tag `<html>` do nosso documento.

Veja o exemplo a seguir:

Hello World AngularJS

```
<html ng-app>
<head>
```

```
<title>Titulo Página</title>
<script
 src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.11/angular.min.js">
</script>
</head>

<body>
    <input ng-model="nome">
    <h1>Olá {{ nome }}</h1>
</body>
</html>
```

Código online: http://plnkr.co/edit/FemetcT1h1HraRQLnT4x

Mesmo este exemplo simples traz à tona algumas características importantes do sistema de templates AngularJS:

- Tags HTML personalizados e atributos: são usados para adicionar comportamento dinâmico de um documento HTML de outra forma estática;
- Chaves duplas (`{{expressão}}`): são utilizadas como um delimitador para expressões que exibem valores do modelo.

No AngularJS, todas as tags HTML e atributos especiais que o framework pode compreender e interpretar referem-se como **diretrizes**. No exemplo, vimos a `ng-app` e a `ng-model`.

Modelo MVC no AngularJS

Quase todas as aplicações web existentes hoje são baseadas de alguma forma no padrão **model-view-controller (MVC)**. Entretanto, o MVC tem um problema: ele não é um padrão muito preciso, mesmo possuindo uma boa arquitetura. Para piorar, existem muitas variações e derivados do padrão original – entre os quais MVP (http://pt.wikipedia.org/wiki/Model-view-presenter) e MVVM (http://en.wikipedia.org/wiki/Model_View_ViewModel) parecem ser os mais populares.

Para aumentar a confusão, há diversas estruturas, e os desenvolvedores tendem a interpretar os padrões mencionados de forma diferente. Isso resulta em situações em que o mesmo nome MVC é utilizado para descrever várias arquiteturas e abordagens de codificação diferentes. Martin Fowler resume isso muito bem em http://martinfowler.com/eaaDev/.

O time do AngularJS tem uma abordagem muito pragmática para toda a família de padrões MVC. Ele declara que o framework é baseado no padrão **MVW (model-view-whatever)**. Basicamente é preciso vê-lo em ação para obter a sensação de sua utilidade:

Visão detalhada

O exemplo que vimos até agora, *"Olá Mundo"*, não empregou qualquer estratégia de estratificação explícita: inicialização de dados e lógica.

Em qualquer aplicação no mundo real, no entanto, é preciso prestar mais atenção a um conjunto de responsabilidades atribuídas a cada camada. Felizmente, o AngularJS oferece diferentes construções arquitetônicas que nos permitem construir adequadamente aplicações mais complexas.

```
<!DOCTYPE html>
<html ng-app="nome_meu_app">
<head>
<meta charset="utf-8">
<title>Título Página</title>
<script
 src="//ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.min.js">
</script>

<script src="app.js"></script>
</head>

<body>
  <div ng-controller="OlaCtrl">
    Como você está <input type="text" ng-model="nome"><br>
    <h1>Eu estou muito bem, {{nome}}! e você?</h1>
    </div>
</body>
</html>
```

Veja que agora usamos uma nova diretiva, `ng-controller`, com uma função JavaScript correspondente. O `OlaCtrl` aceita um argumento `$scope`.

```
var app = angular.module('nome_meu_app', []);

app.controller('OlaCtrl', function($scope) {
```

```
  $scope.nome = 'Giovanni';
});
```

Código online: http://embed.plnkr.co/82gR2y7Z4mOPwz7gEgSG/preview

Scope

Um `$scope`, objeto no AngularJS, está aqui para expor o modelo de domínio para uma visão. Ao atribuir propriedades para uma instância de escopo, podemos fazer novos valores à disposição de um modelo para renderização.

`$scope` pode ser ampliado com dados e funcionalidades específicas para um determinado ponto de vista. Podemos expor lógica específica de UI para os modelos de definição de funções em uma instância de escopo.

Pode-se criar uma função `lerNome` para nossa variável `nome`.

Exemplo:

```
var OlaCtrl = function ($scope) {
   $scope.lerNome = function() {
         return $scope.nome;
    };
};
```

Podemos usá-lo da seguinte maneira:

```
<h1>Olá, {{ lerNome() }}!</h1>
```

O objeto `$scope` nos permite controlar com precisão em qual parte do modelo as operações estão disponíveis para a camada de visão. Conceitualmente, escopos AngularJS estão muito perto do *ViewModel*, o padrão *MVVM*.

Hierarquias de escopo

Vamos olhar de outra forma para o exemplo simples `OlaCtrl` que já fizemos:

```
var OlaCtrl = function ($scope) {
  $scope.nome = 'World';
};
```

O `OlaCtrl` é semelhante a uma função regular do construtor JavaScript, não há absolutamente nada em especial sobre ele além do argumento `$scope`.

Um novo `$scope` foi criado pela diretiva `ng-controller` usando o método `Scope.$New()`. Espere um momento, parece que temos de ter pelo menos uma instância de um escopo para criar um novo escopo. De fato! AngularJS tem uma notação do `$rootScope` (um escopo que é pai de todos os outros escopos). A instância `$rootScope` é criada quando uma nova aplicação é *bootstrapped* (inicializada).

A diretiva `ng-controller` é um exemplo de uma diretiva de criação de escopo. AngularJS irá criar uma nova instância da classe `Scope` sempre que encontrar uma diretiva de criação de escopo na árvore DOM. Um escopo recém-criado irá apontar para o seu pai, usando a propriedade `$parent`.

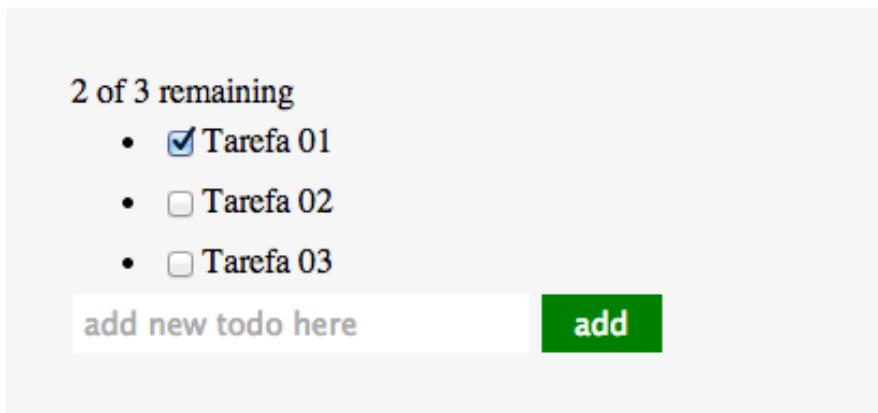
Vamos dar uma olhada no exemplo que faz uso de uma diretiva `ng-repeat`:

```
var MundoCtrl = function ($scope) {
   $scope.populacao = 70000000;
   $scope.paises = [
          {nome: 'Brasil', populacao: 63.1},
          {nome: 'Alemanhã', populacao: 60.1}
  ];
};
```

Nosso HTML fica assim:

```
<ul ng-controller="MundoCtrl">
  <li ng-repeat="pais in paises">
    {{pais.nome}} tem uma população de {{pais.populacao}}
  </li>
  <hr>
  População do mundo: {{populacao}} Milhões de pessoas
</ul>
```

A diretiva `ng-repeat` nos permite iterar sobre uma coleção de objetos (países) e criar novos elementos DOM para cada item em uma coleção. A sintaxe da diretiva deve ser fácil de seguir. Um nova variável `pais` é criada para cada item e exposta em um `$scope`.

Há um problema aqui, ou seja, uma nova variável precisa ser exposta em um `$scope` de cada objeto (`pais`) e não podemos simplesmente substituir valores anteriormente expostos. AngularJS resolve este problema através da criação de um novo escopo para cada elemento de uma coleção.

Views

Nós já vimos exemplos suficientes de modelos AngularJS para perceber que não é mais uma linguagem de modelagem, mas completamente um monstro. Não só pela sua sintaxe de modelo e por nos permitir ampliar o vocabulário HTML, mas também tem a capacidade única de atualizar partes da tela, sem qualquer intervenção manual.

Na realidade, o AngularJS tem até mesmo conexões mais íntimas com o HTML e o DOM, pois depende de um navegador para analisar o texto do modelo. Depois que o navegador tansforma o texto da marcação na árvore DOM, o AngularJS retrocede e percorre a estrutura DOM analisada. Cada vez que encontrar uma diretiva, o AngularJS executa sua lógica para transformar diretivas em partes dinâmicas da tela.

Declarando uma view template

Vamos imaginar que precisamos criar uma *Lista de Tarefas*, cujo propósito é adicionar itens a uma lista, com a opção de marcar como "feito", e que mostre quantas tarefas estão prontas. Para isso, precisamos do seguinte template:

```
<div ng-controller="TarefasCtrl" class="todo">
  <span>{{restante()}} de {{todos.length}} restante</span>

  <ul class="unstyled">
    <li ng-repeat="todo in todos">
      <input type="checkbox" ng-model="todo.feito">
      <span class="feito-{{todo.feito}}">{{todo.text}}</span>
```

```
    </li>
  </ul>

  <form ng-submit="addTodo()">
    <input type="text" ng-model="todoText" size="30"
           placeholder="Adicionar novas tarefas">
    <input class="btn" type="submit" value="adicionar">
  </form>
</div>
```

Vamos trabalhar em cima do código anterior. Em primeiro lugar, é preciso mostrar quantas tarefas prontas eu tenho em minha lista.

```
$scope.restante = function() {
  var count = 0;
  angular.forEach($scope.todos, function(todo) {
    count += todo.feito ? 0 : 1;
  });
  return count;
};
```

A função `restante` é definida pelo controller `TarefasCtrl`. Veja o exemplo a seguir:

```
'use strict';

var ColetaneaFrontEnd = angular.module('meu_app');
ColetaneaFrontEnd.controller('TarefasCtrl', function( $scope ) {

  $scope.restante = function() {
    var count = 0;
    angular.forEach($scope.todos, function(todo) {
      count += todo.feito ? 0 : 1;
    });
    return count;
  };

});
```

Agora, é preciso adicionar elementos em nossa lista para que nossa função `restante()` funcione. Para isso, vou criar uma lista e depois irei manipulá-la com a função `addTodo()` que permite adicionar novos itens.

```
var app = angular.module('meu_app', []);

app.controller('TarefasCtrl', function($scope) {
  //Lista com as tarefas
  $scope.todos = [
    {
      tarefa:'Tarefa 01',
      feito:true
    },
    {
      tarefa:'Tarefa 02',
      feito:false
    },
    {
      tarefa:'Tarefa 03',
      feito:false
    }
  ];

  //Adicionar novos elementos a lista
  $scope.addTodo = function() {
    $scope.todos.push({ tarefa:$scope.todoText, feito:false });
    $scope.todoText = '';
  };

  //Calcular quantas tarefas estão prontas
  $scope.restante = function() {
    var count = 0;
    angular.forEach($scope.todos, function(todo) {
      count += todo.done ? 0 : 1;
    });
    return count;
  };
});
```

Código online: http://embed.plnkr.co/fzc1hHQo8prTgOxKPapD/preview

5.3 Módulos e injeção de dependências

Os leitores atentos provavelmente notaram que alguns dos exemplos apresentados até agora estavam usando funções construtoras globais para definir controllers. Mas

o estado global é o mal da aplicação, já que torna o código difícil de manter, testar e ler. De modo algum AngularJS vai sugerir estado global. Pelo contrário, ele vem com um conjunto de APIs que se torna fácil de definir e que simplifica o registro de objetos no módulo.

Vejamos um controller definido globalmente de forma incorreta.

```
var OlaCtrl = function ($scope) {
  $scope.nome = 'World';
}
```

E agora da forma correta, como AngularJS indica a fazer.

```
angular
  .module('hello', [])
  .controller('OlaCtrl', function($scope) {
    $scope.nome = 'World';
  });
```

AngularJS define o namespace `angular` global. Existem várias funções de utilidade e conveniência expostas neste namespace, o módulo é uma dessas funções. Um módulo atua como um contêiner para outros objetos gerenciados (controllers, serviços e assim por diante). Há muito mais a aprender sobre os módulos do que namespacing simples e organização de código.

Para definir um novo módulo, precisamos fornecer o seu nome como o primeiro argumento para a chamada de função módulo. O segundo argumento torna possível expressar uma dependência de outros módulos.

A chamada para a função `angular.module` retorna uma instância de um módulo recém-criado. Assim, como nós temos acesso a este exemplo, podemos começar a definir novos controllers.

Injeção de dependências

O termo injeção de dependência foi cunhado por Martin Fowler (http://martinfowler.com/articles/injection.html) ; em linhas gerais, é uma técnica para prover dependências externas a um determinado componente de software. Temos algumas vantagens:

1) Simplificação no código da aplicação: com a injeção de dependência, os objetos da aplicação são liberados da tarefa de trazer suas próprias dependências –

redução de código para inicialização e configuração de dependências –, eles ficam livres para executar apenas suas regras de negócio, pois sabem que as suas dependências estarão lá quando necessárias.

2) Testabilidade: se as dependências podem ser injetadas em um componente, torna-se possível (e talvez até mais fácil) injetar implementações *mocks* dessas dependências.

3) Baixo acoplamento entre os objetos: o objeto conhece suas dependências apenas por sua interface pública – não por sua implementação, nem por como foram instanciadas. Assim, a dependência pode ser trocada por uma implementação diferente, sem que o objeto dependente conheça a diferença.

4) Flexibilidade no gerenciamento do ciclo de vida dos objetos: objetos podem ser cacheados, ser um *singleton* ou ter uma vida curta – tudo controlado por meio de configurações e/ou pelo contêiner.

Vejamos um exemplo:

```
function Filmes(sobreFilme) {
  this.sobreFilme = sobreFilme;
}

Filmes.prototype.mudaNome = function(nome) {
  this.sobreFilme.exibir(nome);
}
```

Repare que `Filmes` não está preocupado com a localização de `sobreFilme`: ele simplesmente é entregue a `sobreFilme` em tempo de execução. Isso é desejável, mas coloca a responsabilidade de se apoderar da dependência sobre o código que constrói `Filmes`.

Para gerenciar a responsabilidade de criação de dependência, cada aplicativo AngularJS tem um *injector*. O *injector* é um localizador de serviço que é responsável pela construção e pesquisa de dependências.

Um exemplo do uso do serviço do *injector*:

```
angular
  .module('MeusFilmes', [])
  .factory('sobreFilme', function($window) {
    return {
```

```
      exibir: function(text) {
        $window.alert(text);
      }
    };
  });

var injector = angular.injector(['MeusFilmes', 'ng']);

var sobreFilme = injector.get('sobreFilme');
```

Requisitar as dependências resolve a questão da codificação difícil, mas significa também que o `injector` precisa ser passado em todo o aplicativo. Passar o `injector` quebra a Lei de Demeter. Para corrigir isso, transformamos a responsabilidade de pesquisa de dependência para o `injector`, declarando as dependências:

A Lei de Demeter foi descrita pela primeira vez na Universidade Northeastern, em Boston, Massachusetts em 1987. Na verdade essa "Lei" fornece um conjunto de orientações úteis para o projeto de software. Quando estas regras são seguidas, reduzimos o acoplamento, descrevendo como os métodos e propriedades das classes devem se comunicar entre si. A lei é por vezes conhecida como o Princípio do Mínimo Conhecimento.
– José Carlos Macoratti

Por exemplo:

```
<div ng-controller="FilmesController">
  <button ng-click="falaFilme()">Nome do Filme</button>
</div>

function FilmesController($scope, sobreFilme) {
  $scope.falaFilme = function() {
    sobreFilme.exibir('Nome do filme é ...');
  };
}

injector.instantiate(FilmesController);
```

Observe que o fato de ter o `ng-controller` instanciando a classe faz com que ele possa satisfazer todas as dependências de `FilmesController`, sem que o controller conheça o injector. Este é o melhor resultado. O código do aplicativo simplesmente declara as dependências que precisa, sem ter que lidar com o injector. Esta configuração não infringe a Lei de Demeter.

5.4 Service AngularJS

Angular nos oferece diversos serviços úteis para qualquer aplicação não trivial que você vai achar útil para próprios serviços personalizados. Para fazer isso, você deve começar por registrar um *service factory* com um módulo de serviço, através do Módulo *#factory API* ou diretamente através da *$provide API* dentro da função de configuração do módulo.

De todos os serviços angular, participar de injeção de dependência (DI) por si só é registrar com o sistema de Angular DI (injector) sob um nome (id), bem como declarando dependências que precisam ser fornecidos para a função de fábrica do serviço registrado. A capacidade de trocar as dependências de objetos *mocks/stubs/dummies* em testes permite que os serviços sejam altamente testáveis.

Serviços criados pelo `$injector` são singletons. Haverá apenas uma instância de um determinado serviço por instância de um aplicativo em execução.

Registrando Service

Para registrar um *service*, você deve ter um módulo do qual este serviço irá fazer parte. Depois, você pode registrar o serviço com o módulo através da *API Module* ou usando o serviço `$provide` na função de configuração do módulo. O seguinte pseudocódigo mostra as duas abordagens:

Usando a API `angular.Module`:

```
var showMensagemModulo = angular.module('showMensagemModulo', []);

showMensagemModulo.factory('serviceId', function() {
  var sumarioNovaInstancia;

  return sumarioNovaInstancia;
});
```

Usando o service `$provide`:

```
angular.module('showMensagemModulo', [], function($provide) {
  $provide.factory('serviceId', function() {
    var sumarioNovaInstancia;

    return sumarioNovaInstancia;
  });
});
```

Note que você não está registrando uma instância de serviço, mas sim uma função `factory` que vai criar essa instância quando chamada.

Dependências

Serviços não só podem ser dependentes, mas também pode ter suas próprias dependências. Estes podem ser especificados como argumentos da função `factory`.

A seguir, veja um exemplo de um serviço muito simples. Ele depende do serviço `$window` (que é passado como um parâmetro para a função `factory`) e é apenas uma função. O serviço simplesmente armazena todas as notificações; após a terceira, ele exibe todas as notificações por alerta:

```
angular.module('showMensagemModulo', [], function($provide) {
  $provide.factory('notifica', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        win.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  }]);
});
```

Um novo recurso do Angular permite determinar a dependência a partir do nome do parâmetro. Vamos reescrever o exemplo anterior para mostrar o uso desta injeção implícita no `$window`, `$scope`, e nosso serviço de notificações.

```
angular.
   module('showMensagemModulo', []).
   factory('notifica', function($window) {
      var msgs = [];
      return function(msg) {
        msgs.push(msg);
        if (msgs.length == 3) {
          $window.alert(msgs.join("\n"));
          msgs = [];
        }
      };
    });
```

```
  function showController($scope, notifica) {
    $scope.notificarChamada = function(msg) {
      notifica(msg);
    };
  }

<div id="implicit" ng-controller="showController">
  <p>(Clique 3 vezes para ver um alerta)</p>
  <input ng-init="mensagem='test'" ng-model="mensagem">
  <button ng-click="notificarChamada(mensagem);">
    exibir notificação
  </button>
</div>
```

Código online: http://plnkr.co/edit/HG1iZBLIvnQnBoTtDEk9

Todos os serviços em Angular são instanciados *lazily*. Isso significa que um serviço será criado somente quando ele é necessário para a instanciação de um serviço ou um componente da aplicação que depende dele. Em outras palavras, Angular não vai instanciar serviços, a menos que sejam solicitados diretamente ou indiretamente pela aplicação. Por último, é importante perceber que todos os serviços Angular são aplicações únicas, ou seja, há apenas um objeto de um determinado serviço por injector.

Componentes do Módulo Angular: http://docs.angularjs.org/api/ng

5.5 Comunicando com servidor back-end

O serviço `$http` é um serviço central Angular que facilita a comunicação com os servidores HTTP remotos via objeto `XMLHttpRequest` do navegador ou através de `JSONP`.

O *$http API* baseia-se nas APIs *deferred/promise* expostas pelo service `$q`. Enquanto que para os padrões de uso simples isso não importa muito, para uso avançado é importante para se familiarizar com essas APIs e as garantias que oferecem.

Uso geral

O serviço `$http` é uma função que recebe um único argumento – um objeto de configuração – que é usado para gerar uma solicitação HTTP e retorna uma promessa com dois métodos `$http` específicos. São eles: `success` e `error`.

```
$http({method: 'GET',
  url: '//api.twitter.com/1.1/search/tweets.json?q=angularjs'}).
  success(function(data, status, headers, config) {
    // Sucesso get
    data // Retorno do objeto feita pela requisição
  }).
  error(function(data, status, headers, config) {
    // Sucesso get
  });
```

Uma vez que o valor retornado da função `$http` é uma `promise`, você também pode usar o método depois de registrar retornos de chamada. E esses retornos de chamada receberão um único argumento – um objeto que representa a resposta.

Um código de status de resposta entre 200 e 299 é considerado um status de sucesso e resultará no retorno sucesso sendo chamado. Note-se que, se a resposta for um redirecionamento, `XMLHttpRequest` vai segui-lo de forma transparente, o que significa que o retorno de erro não será chamado para tais respostas.

Métodos curtos

Todas as invocações ao serviço `$http` exigem um método HTTP e a URL a ser chamada. Para simplificar, foram criados métodos de atalho direto com o nome do método HTTP, como `get` e `post`:

```
$http.get('//api.twitter.com/1.1/search/tweets.json?q=angularjs')
  .success(successCallback);

$http.post('//api.twitter.com/1.1/search/tweets.json?q=angularjs', data)
  .success(successCallback);
```

Veja um exemplo real de request `$http`:

```
<body ng-controller="PessoasCtrl">
  <p ng-repeat="dados in pessoas">
    Oi, meu nome é <strong>{{ dados.nome }}</strong>
    tenho <strong>{{ dados.idade }}</strong> anos de idade,
    moro em <strong>{{ dados.cidade }}</strong>!
  </p>
</body>

var app = angular.module('plunker', []);
```

```
app.controller('PessoasCtrl', function($scope, $http) {
  $http.get('dados.json').success(function(data) {
    $scope.pessoas = data;
  });
});
```

```
[
    {
        "idade": 23,
        "cidade": "Belo Horizonte",
        "nome": "Giovanni Keppelen"
    },
    {
        "idade": 25,
        "cidade": "Rio de Janeiro",
        "nome": "Keppelen"
    }
]
```

Código online: http://plnkr.co/edit/aHxoFbhGoATluQNrqfRJ?p=preview

Para estudos, os outros métodos do Angular para Ajax:

1) $http.get => http://docs.angularjs. org/api/ng/service/$http#get

2) $http.head => http://docs.angularjs. org/api/ng/service/$http#head

3) $http.post => http://docs.angularjs. org/api/ng/service/$http#post

4) $http.put => http://docs.angularjs. org/api/ng/service/$http#put

5) $http.delete => http://docs.angularjs. org/api/ng/service/$http#delete

6) $http.jsonp => http://docs.angularjs. org/api/ng/service/$http#jsonp

5.6 $ROUTE ANGULARJS

AngularJS tem um `$route service`, que pode ser configurado para lidar com transições de rotas em aplicações web de uma única página. Abrange todas as características do `$location` e, adicionalmente, oferece outras facilidades muito úteis.

O serviço `$location` analisa a URL do navegador (com base no `window.location`) e faz a URL disponível para sua aplicação. Mudanças na

URL são refletidas no serviço `$location` e as alterações em `$location` são refletidas na barra de endereços.

A partir da versão 1.2, o AngularJS teve seu sistema de roteamento entregue em um arquivo separado (`angular-route.js`) com o seu próprio módulo (`ngRoute`). Lembre-se de incluir o arquivo `-angular route.js` e declarar uma dependência no módulo `ngRoute`.

Rotas básicas

Em AngularJS, rotas podem ser definidas durante a configuração do aplicativo usando o serviço `$routeProvider`. A sintaxe usada pelo serviço `$routeProvider` é semelhante a usada com `$location` personalizado. Veja um exemplo:

```
angular.module('ExemploNgRoute', ['ngRoute'])
       .config(function($routeProvider, $locationProvider) {

  $routeProvider.when('/livro/:livroId', {
    templateUrl: 'livro.html',
    controller: LivroCtrl,
    resolve: {
      delay: function($q, $timeout) {
        var delay = $q.defer();
        $timeout(delay.resolve, 1000);
        return delay.promise;
      }
    }
  });

  $routeProvider.when('/livro/:livroId/ch/:capituloId', {
    templateUrl: 'capitulo.html',
    controller: CapituloCtrl
  });

  $locationProvider.html5Mode(true);
});
```

O serviço `$routeProvider` expõe uma API de estilo fluente, na qual passamos o método que encadeia chamadas para a definição de novas rotas e da criação de uma rota default.

Uma vez iniciado, o pedido não pode ser reconfigurado para suportar rotas adicionais (ou remover as já existentes). Isso está ligado ao fato de que os providers Angular podem ser injetados e manipulados somente em blocos de configuração executados durante inicialização do aplicativo.

No exemplo anterior, você pode ver que `TemplateUrl` era a única propriedade para cada rota, mas a sintaxe oferecida pelo serviço `$routeProvider` é muito mais rica.

O conteúdo de uma rota pode também ser especificada em linha usando a propriedade modelo de um objeto de definição de rota. Embora esta seja uma sintaxe suportada, codificar marcação da rota em sua definição não é muito flexível (nem sustentável), por isso raramente é utilizada na prática.

Exemplo mais completo de `$route`:

`Index.html`

```
<div ng-app="ExemploNgRoute">
<div ng-controller="MeusLivrosCtrl" class="tabs">
  <a href="Livro/Angular">Angular</a>
  <a href="Livro/Angular/ch/1">Angular: Ch1</a>
  <a href="Livro/jQuery">jQuery</a>
  <a href="Livro/jQuery/ch/4?key=value">jQuery: Ch4</a>
  <a href="Livro/VanilhaJS">VanilhaJS</a><br/>

  <div ng-view></div>
  <hr />

  <pre>$location.path() = {{$location.path()}}</pre>
  <pre>$route.current.templateUrl = {{$route.current.templateUrl}}</pre>
  <pre>$route.current.params = {{$route.current.params}}</pre>
  <pre>$route.current.scope.name = {{$route.current.scope.name}}</pre>
  <pre>$routeParams = {{$routeParams}}</pre>
</div>
</div>
```

livro.html

```
controller: {{nome}}<br />
Livro Id: {{params.livroId}}<br />
```

capitulo.html

```
controller: {{name}}<br />
Livro Id: {{params.bookId}}<br />
Capitulo Id: {{params.chapterId}}
```

script.js

```
angular.module('ExemploNgRoute', ['ngRoute'])
.config(function($routeProvider, $locationProvider) {
  $routeProvider.when('/Livro/:livroId', {
    templateUrl: 'livro.html',
    controller: livroCtrl,
    resolve: {
      // Com 1 segundo de atraso
      delay: function($q, $timeout) {
        var delay = $q.defer();
        $timeout(delay.resolve, 1000);
        return delay.promise;
      }
    }
  });
  $routeProvider.when('/Livro/:livroId/ch/:capituloId', {
    templateUrl: 'capitulo.html',
    controller: capituloCtrl
  });

  // Para configurar os inks dentro do plunker
  $locationProvider.html5Mode(true);
});

function MeusLivrosCtrl($scope, $route, $routeParams, $location) {
  $scope.$route = $route;
  $scope.$location = $location;
  $scope.$routeParams = $routeParams;
}

function livroCtrl($scope, $routeParams) {
  $scope.nome = "livroCtrl";
  $scope.params = $routeParams;
}

function capituloCtrl($scope, $routeParams) {
  $scope.nome = "capituloCtrl";
```

```
  $scope.params = $routeParams;
}
```

Código online: http://embed.plnkr.co/8zNXCmheCPCsnxqr78YH/preview

5.7 Conclusão

Comunidade

Todos sabem que projeto nenhum vai para frente se não tem pessoas por trás. E AngularJS é open source, apesar de ter todo um time do Google trabalhando diariamente nele. Há também uma comunidade muito grande e com muitas opções de aprendizado:

1) angular@googlegroups.com (Lista email no Google)

2) https://plus.google.com/u/0/communities/115368820700870330756 (Comunidade no Google+)

3) http://stackoverflow.com (starckoverflow)

4) https://www.facebook.com/groups/angularjsbrasil/ (Comunidade Facebook)

5) http://ng-learn.org/

6) #angularjs IRC channel

Recurso online

AngularJS tem seu próprio website (http://www.angularjs.org) , onde podemos encontrar tudo, absolutamente tudo o que precisar: visão geral conceitual, tutoriais, guia do desenvolvedor, referência da API, e assim por diante. O código-fonte para todas as versões AngularJS liberados pode ser baixado em http://code.angularjs.org.

As pessoas que procuram exemplos de código não irão se decepcionar, pois o AngularJS tem sua própria documentação e tem bastante trechos de código. Além de tudo isso, pode-se navegar em uma galeria de aplicações construídas com AngularJS (http://builtwith.angularjs.org) . Um canal do YouTube dedicado a isso é http://www.youtube.com/user/angularjs.

Depoimentos

"Nós tínhamos acabado de lançar um e-commerce, que poucos meses depois se tornou o segundo maior do ramo esportivo da América Latina. Usamos jQuery e uma centena de plugins. O código? Era uma sopa de seletores jQuery, event handlers e callbacks. Dar manutenção, ou seja, mudar o código era realmente um sofrimento. Decidimos reescrever a aplicação usando o AngularJS, bem antes do hype, pela testabilidade do JavaScript. Além de refazer a aplicação inteira em apenas 2 semanas (contando com a reescrita de todo o CSS), o resultado foi um código fácil de mudar, fácil de se trabalhar em equipe e fácil de entender. E o melhor de tudo, aprendemos boas práticas de Desenvolvimento de Software, graças ao Design do framework"
– Ciro Nunes (Front-end Engineer Avenue Code)

Obrigado

AnglarJS tem muito mais potencial do que mostrei a vocês. O que escrevi foi apenas overview do que AngularJS é capaz. Poderia escrever páginas e mais páginas sobre coisas legais e funcionalidades sobre ele. Acredito que pude tirar muitas dúvidas de quem não o conhecia ou tinha um breve conhecimento sobre AngularJS.

Obrigado pela companhia de todos e espero que tenham gostado.

Sobre o autor

Giovanni Keppelen é CTO na Planedia, plataforma online para planejar e planificar sua viagem. Cofundador da BrazilJS Foundation, idealizador e organizador do Front in BH e Rio.JS. Um dos autores do Browser Diet, com Zeno Rocha, Sérgio Lopes, Marcel Duram (Twitter), Renato Mangini (Google) entre outros grandes nomes. Trabalhou como Front-end Engineer no Peixe Urbano e coordenador Front-end na Mobicare à frente de grandes projetos. Cursando Produção Multimídia no UNI-BH.

Capítulo 6

As diretrizes de acessibilidade para conteúdo na Web – WCAG

6.1 Acessibilidade na Web

Acessibilidade na Web significa garantir que pessoas com deficiência devem poder usar, acessar e compreender o que é publicado na Web. Considerar o uso de diretrizes internacionais de acessibilidade beneficia um grande grupo de pessoas:

- pessoas cegas, que utilizam um software leitor de tela para acessar o conteúdo de uma página;
- pessoas com baixa visão, que necessitam de contraste e fontes maiores;
- pessoas daltônicas que não enxergam um determinado tipo de cor;
- pessoas surdas que não conseguem escutar um áudio na Web;

- pessoas com mobilidade reduzida que utilizam um dedo (ou nem isso) para acessar o conteúdo de uma página Web.

Porém, acessibilidade na Web vai muito além de beneficiar somente pessoas com deficiências. Usuários de dispositivos móveis, pessoas com deficiência temporária e usuários idosos são apenas alguns dos diversos exemplos de usuários que podem ter seu acesso a Web prejudicado se as páginas não forem acessíveis.

Segundo o Censo do IBGE de 2010, o Brasil conta com aproximadamente 24% da população com algum tipo de deficiência. São mais de 45 milhões de pessoas que podem ter alguma barreira de acesso caso as páginas não sejam projetadas de forma a atender todas as pessoas. O princípio do desenho universal é fundamental para que todos tenham acesso pleno ao conteúdo disponível na Web. Para isso foram criadas as Diretrizes Internacionais de Acessibilidade na Web.

Dados do censo do IBGE

O censo de 2010 traz dados interessantíssimos sobre acessibilidade e deficiência:

http://censo2010.ibge.gov.br/noticias-censo?busca=1&id=3&idnoticia=2170&view=noticia

6.2 Por trás do WCAG 2.0

A Web nasceu para ser acessível. Em 1997, Tim Berners-Lee anunciava o lançamento do *International Program Office* (IPO) para a Iniciativa de Acessibilidade na Web – WAI (http://www.w3.org/WAI/) com o objetivo de promover e garantir o funcionamento da Web para pessoas com deficiências. Durante seu lançamento, o criador da Web declarou:

"O poder da Web está em sua universalidade. Acesso por todos, independentemente da deficiência, é um aspecto essencial"
– Tim Berners-Lee em http://www.w3. org/Press/IPO-announce

Esse programa alavancou diversas iniciativas e projetos relacionados à acessibilidade na Web. Um dos mais importantes foi a criação das Diretrizes de Acessibilidade para Conteúdo na Web – as WCAG 1.0 (http://www.w3.org/TR/WCAG10/)

, publicadas como recomendação do W3C no dia 5 de maio em 1999, com o objetivo de orientar como tornar o conteúdo acessível para pessoas com deficiências. O documento tem uma ótima introdução, que explica que, além de promover a acessibilidade para pessoas com deficiências, seguir as diretrizes beneficia todas as pessoas em diferentes situações, como acessar uma página com iluminação fraca, ou em um ambiente com muito barulho ou com as mãos ocupadas.

Nove anos depois, com a Web em ampla evolução, a WAI lança a segunda versão das Diretrizes de Acessibilidade, as WCAG 2.0 (http://www.w3.org/TR/WCAG/) . Baseadas na primeira versão da recomendação, o documento muda sua estrutura e amplia as formas e técnicas para tornar o conteúdo acessível.

A nova versão da recomendação está organizada da seguinte forma:

- **Princípios** – são quatro princípios que fornecem a base para a acessibilidade da Web: perceptível, operável, compreensível e robusto.
- **Diretrizes** – as 12 diretrizes sob os princípios fornecem os objetivos básicos que devem ser trabalhados para que o conteúdo na Web seja acessível às pessoas com deficiências.
- **Critérios de sucesso** – para cada diretriz, existem critérios de sucesso para atingir o objetivo de cumprir as diretrizes e garantir que as WCAG 2.0 sejam efetivamente usadas em especificações de projetos em geral, como testes de conformidade etc. Os critérios de sucesso são divididos em três níveis de conformidade: A (mais baixo), AA, AAA e (mais alto).
- **Técnicas de tipo suficiente e aconselhadas** – para cada uma das diretrizes e critérios de sucesso no documento WCAG 2.0, existe uma grande variedade de técnicas. As técnicas são divididas em duas categorias: suficiente e aconselhadas.

  Técnicas *suficientes* garantem o cumprimento dos critérios de sucesso, apontando documentação técnica necessária para atingir os objetivos. Já as técnicas *aconselhadas* vão além do que é exigido nas técnicas suficiente e abrangem uma ampla gama de barreiras de acessibilidade que podem não ser abordadas pelas técnicas suficientes.

O objetivo deste artigo é apresentar os quatro princípios das WCAG 2.0, boa parte das diretrizes e quais as técnicas para melhorar a acessibilidade do seu projeto Web. Não vou passar detalhadamente por cada uma delas, mas gostaria de explicar,

pelo menos boa parte dos principais critérios de sucesso relacionados no nível A de forma simples e descomplicada. Boa parte dos exemplos de código foram baseados em exemplos da própria documentação do W3C.

É importante ressaltar que apenas seguir o que este artigo orienta não garante que sua página seja totalmente acessível. A principal razão de descrever a documentação do W3C desta forma é tornar mais fácil a compreensão e quebra de barreiras mais comuns para melhorar a acessibilidade das páginas Web. Você vai notar que algumas técnicas não estão relacionadas a desenvolvimento de código, mas em boas práticas e alternativas para tornar o conteúdo da Web acessível a todos.

6.3 Princípio 1: perceptível

A informação e os componentes da interface do usuário têm de ser apresentados aos usuários em formas que eles possam perceber.

O que isso significa? Que os usuários devem ser capazes de perceber a informação que está sendo apresentada, não podendo ser invisível para todos os seus sentidos.

Diretriz 1.1 – alternativas em texto

Todo o conteúdo não textual deve fornecer alternativas em texto. Assim, ele pode ser transformado conforme a necessidade do usuário, como em fontes maiores, braile, sons etc. Essa diretriz abrange uma ampla gama de situações:

1.1.1 Conteúdo não textual

Quando o conteúdo não textual é apresentado ao usuário, ele deve ter uma alternativa em texto que serve para a finalidade equivalente. Esse critério de sucesso não se aplica às seguintes situações: controles de formulário, mídia baseada em tempo, testes, experiências sensoriais, captcha e imagens decorativas.

São definidas algumas situações de uso:

Situação A – a descrição curta serve ao mesmo propósito e apresenta a mesma informação do conteúdo não textual.

Exemplo: um documento HTML com uma foto da estátua do Cristo Redentor.

Considerando que esse elemento não textual é uma foto, a forma para solucionar essa situação é simples. Basta utilizar um elemento de imagem com o atributo `alt` preenchido de forma adequada:

```
<img src="foto.png" alt="Foto do Cristo Redentor">
```

Com isso, você garante que tecnologias assistivas consigam ler o significado da imagem pelo seu texto alternativo.

A parte difícil não é colocar o atributo `alt` nas imagens e sim, como descrevê-la. A documentação sugere fazer as seguintes perguntas para entender a necessidade e o que descrever no atributo `alt`.

- Por que este conteúdo não textual está aqui?
- Quais as informações são apresentadas?
- Qual o objetivo ele deve cumprir?
- Se eu não puder usar o conteúdo não textual, que palavras eu iria usar para transmitir a mesma função e/ou informação?

É importante ressaltar que, além de tecnologias assistivas, ferramentas de busca indexam a informação dentro do atributo `alt`.

Situação B – a descrição curta não serve ao mesmo propósito e apresenta a mesma informação do conteúdo não textual.

Exemplo: um gráfico ou diagrama detalhado.

Nesse caso, as WCAG recomendam, além de um texto alternativo curto explicando brevemente o gráfico ou diagrama, duas possibilidades: colocar um link próximo à imagem que vá para uma página ou arquivo com a descrição completa ou colocar um texto na própria página, próximo ao gráfico ou diagrama, que explica detalhadamente imagem.

Essa técnica garante que uma imagem ou diagrama tenha seu conteúdo convertido em modo texto. Assim, ferramentas de busca, navegadores e tecnologias assistivas podem acessar as informações de forma plena.

Situação C – o elemento não textual é um campo de formulário.

Um campo de formulário não tem uma alternativa em texto, mas tem rótulos que definem seu propósito e o tipo de dado que deve ser inserido. Nesse caso, a recomendação é fazer a relação adequada dos atributos IDs dos formulários com seus respectivos rótulos, utilizando o elemento `<label>`.

```
<label for="email">E-mail</label>
<input type="text" name="email" id="email">
```

Se não for possível utilizar o elemento `label`, utilize o atributo `title` para identificar os campos de formulário.

```
<input id="area" name="area" title="Código de área" type="text">
<input id="primeiros" name="primeiros" type="text"
      title="Primeiros quarto dígitos do número do telefone">
<input id="ultimos" name="ultimos" type="text"
      title="Últimos quarto dígitos do número de telefone">
```

Utilize esta técnica somente se não for possível usar o elemento `label` nos campos do formulário.

No caso de botões com `<input type="img">`, utilize o atributo `alt` para indicar a ação a ser feita; por exemplo, `alt="enviar formulário"`.

Essa técnica garante que tecnologias assistivas possam relacionar adequadamente os respectivos textos com seu campo de formulário. Além da melhoria na acessibilidade, o formulário tem um ganho enorme em usabilidade, já que campos radiobox e checkbox que relacionam `label` com IDs tornam o texto do rótulo um objeto selecionável, não sendo mais necessário clicar somente no quadrado/bolinha do campo. O texto também aciona o campo.

Situação D – o elemento não textual é uma mídia baseada em tempo.

A descrição detalhada será feita na diretriz 1.2.

Situação E – o elemento não textual é um *CAPTCHA*.

CAPTCHA é a abreviação de *Completely Automated Public Turing test to tell Computers and Humans Apart* que representa um desafio, normalmente em imagem, para evitar que robôs preencham automaticamente um formulário. O problema é que essas imagens são inacessíveis para usuários cegos, que navegam com softwares leitores de tela. Caso seja inevitável o uso do CAPTCHA, a recomendação é utilizar um texto curto para descrevê-lo e utilizar outra modalidade de CAPTCHA que não seja invisível para o usuário. Por exemplo, usar um CAPTCHA com alternativa em áudio, ou utilizar perguntas simples, em modo texto na página.

Situação F – o elemento não textual deve ser ignorado pelas tecnologias assistivas. Caso o elemento não textual deva ser ignorado pelas tecnologias assistivas, como uma imagem decorativa, ela deve ser inserida dentro das CSS.

```
<style>
    body {
        background: #ffe url('/images/home-bg.jpg') repeat;
    }
</style>
```

Diretriz 1.2 – mídias com base em tempo

Garantir que pessoas com deficiência tenham acesso a elementos de mídia, como áudio e vídeo é o objetivo desta diretriz. Para áudio e vídeo pré-gravado, os critérios de sucesso para atender a essa diretriz são os seguintes:

1.2.1 Apenas áudio e apenas vídeo

Para áudio e vídeo pré-gravados na Web, ofereça uma alternativa para esse tipo de mídia, como a transcrição do áudio/vídeo em um documento texto ou outra página Web. Essa técnica beneficia pessoas que não conseguem ouvir um áudio na web e também oferece mais conteúdo a ser indexado por ferramentas de busca.

1.2.2 Legendas (pré-gravadas)

A documentação do WCAG orienta a fornecer legendas para conteúdo gravado, no formato *open caption* (sempre disponível) ou *closed caption* (que o usuário pode habilitar ou não). As WCAG não orientam sobre um formato específico de legendas, mas mostram diversas opções de tecnologias que possibilitam essa ação.

1.2.3 Audiodescrição ou mídia alternativa (pré-gravada)

Audiodescrição é uma trilha de áudio que fornece informações que compreendemos visualmente e que não estão contidas em diálogos de um vídeo. A audiodescrição tem um impacto enorme em quem assiste a um vídeo, mas não consegue ver o que é exibido. No Brasil, os canais de TV aberta são obrigados a exibir duas horas de programação semanal com audiodescrição. Faça um teste e assista a um filme com esse recurso e perceba o quanto ele é importante para quem não enxerga o que está sendo exibido.

Diretriz 1.3 – adaptável

Esta diretriz orienta o desenvolvedor a criar conteúdos que possam ser adaptados ou apresentados em diferentes formas (como em um layout mais simples) sem que ele perca sua estrutura ou informação contida nele.

1.3.1 Informações e relações

É preciso garantir que as informações, a estrutura e as relações que são transmitidas através da página ou da aplicação Web estejam disponíveis de forma progra-

mática ou estejam disponíveis no texto. Alguns exemplos para ilustrar esse critério são:

- Um formulário com campos obrigatórios marcados adequadamente;
- Um formulário que usa cor e texto para marcar campos obrigatórios;
- Uma tabela de horário de ônibus que tem seus cabeçalhos definidos programaticamente.

Existem diversas técnicas que vão de encontro a essa orientação. Algumas delas são as seguintes:

Utilize elementos semânticos para marcar a estrutura.

A regra aqui é garantir que os elementos tenham sua semântica correta apresentada. A semântica facilita o uso de tecnologias assistivas, pois a grande maioria dos usuários utiliza atalhos de teclado para navegar. Dessa forma, fica mais fácil navegar por cabeçalhos, tabelas, formulários etc.

Utilizar tabelas para marcar conteúdo tabular.

Historicamente, tabelas foram utilizadas para a criação de layout de páginas. Essa técnica é desencorajada, pois causa problemas a usuários de tecnologias assistivas. Para a exibição de dados tabulares, que é realmente sua função, existem algumas orientações:

Utilizar o elemento CAPTION para o título da tabela.

É mais semântico utilizar o elemento `caption` que está dentro da tabela, pois assim o usuário não precisa sair e procurar o título referente a ela. Utilize o elemento da seguinte forma:

```
<table>
  <caption>Agenda da semana</caption>
  ...
</table>
```

O elemento `caption` pode ter seu posicionamento e formatação modificados por CSS.

Utilize o atributo `scope` para classificar linhas e colunas de tabelas complexas. Utilize os valores `row`, `col`, `rowgroup` e `colgroup`. Isso facilita a orientação de tecnologias assistivas dentro de uma tabela.

```
<table border="1">
  <caption>Dados de contato</caption>
  <tr>
    <td></td>
    <th scope="col">Nome</th>
    <th scope="col">Telefone</th>
    <th scope="col">Fax</th>
    <th scope="col">Cidade</th>
  </tr><tr>
    <td>1.</td>
    <th scope="row">João dos Santos</th>
    <td>999-999-9999</td>
    <td>999-999-9999</td>
    <td>Belo Horizonte</td>
  </tr><tr>
    <td>2.</td>
    <th scope="row">Carla Pereira</th>
    <td>999-999-9999</td>
    <td>999-999-9999</td>
    <td>São Paulo</td>
  </tr><tr>
    <td>3.</td>
    <th scope="row">Maria Souza</th>
    <td>999-999-9999</td>
    <td>999-999-9999</td>
    <td>Rio de Janeiro</td>
  </tr>
</table>
```

Para tabelas mais complexas, é possível utilizar Ids e Headers para uma relação melhor entre suas células, mas a recomendação mais importante é deixar as tabelas simples. É muito mais fácil compreender os dados quando as tabelas são de fácil leitura.

Tabelas com Ids e Headers

Caso tenha interesse nessa técnica, leia *"Using id and headers attributes to associate data cells with header cells in data tables*", disponível em http://www.w3.org/TR/2013/NOTE-WCAG20-TECHS-20130905/H43

Utilize a H1-H6 para estruturar cabeçalhos.

Essa técnica garante que a semântica dos cabeçalhos seja preservada e que usuários de tecnologias assistivas consigam navegar por eles. As WCAG sugerem dois exemplos de abordagem:

Organização hierárquica de cabeçalhos:

```
<h1>Produtos em destaque<h1>
...
        <h2>Eletrônicos</h2>
...
                <h3>Televisores</h3>
...
        <h2>Alimentação</h3>
...
                <h3>Frutas</h3>
```

Cabeçahos em layout:

```
<head>
 <title>Mercado de ações hoje</title>
 </head>

 <body>

 <!-- left nav -->
 <div class="left-nav">
 <h2>Navegação</h2>
 <!-- content here -->
 </div>

 <!-- main contents -->
 <div class="main">
 <h1>Mercado de ações hoje</h1>
 <!-- article text here -->
 </div>

 <!-- right panel -->
 <div class="left-nav">
 <h2>Links relacionados</h2>
 <!-- content here -->
 </div>
 </body>
```

É importante ressaltar que esse exemplo de código não é uma regra de como estruturar cabeçalhos dentro de um layout de página. Cada desenvolvedor pode estruturar da forma que achar melhor, desde que siga a lógica e o critério de importância de cada nível de cabeçalho.

Forneça descrição adequada para grupos de formulários utilizando os elementos LEGEND e FIELDSET.

É recomendável agrupar blocos de formulário conforme o tipo de dado solicitado. Muitas vezes, as páginas com formulários pedem dados pessoais, dados comerciais etc. Para evitar que um campo *nome* ou *telefone* apareça duas vezes sem sua devida marcação de grupo, utilize o elemento `fieldset` para agrupar os blocos e o elemento `legend` para declarar o título adequado daquele grupo.

```
<fieldset>
  <legend>Dados pessoais</legend>
      <label for="nome">Nome</label>
    <input type="text" name="nome" id="nome">
    <label for="end">Endereço</label>
    <input type="text" name="end" id="end">
...
</fieldset>
<fieldset>
  <legend>Dados Comerciais</legend>
      <label for="nome-com">Nome da empresa</label>
    <input type="text" name="nome-com" id="nome-com">
    <label for="end-com">Endereço</label>
    <input type="text" name="end-com" id="end-com">
...
</fieldset>
```

1.3.2 Sequência com significado

Oferecer ao usuário uma sequência de leitura correta caso a sequência do conteúdo afete seu significado. Isso é muito comum quando temos uma página com múltiplas colunas ou quando acessamos websites de outros idiomas, cuja sequência de leitura é da direita para a esquerda. Nesse caso, além de ordenar o conteúdo de forma estruturada, utilizar atributos que programaticamente oriente a leitura adequada do conteúdo é recomendável, como o RLM (right-to-left mark) e LMR (left-to-right mark).

1.3.3 Características sensoriais

Não fazer um conteúdo que dependa exclusivamente de características sensoriais dos seus componentes, como forma, tamanho, localização visual, orientação ou som. O exemplo do uso da cor (critério de sucesso 1.4.1) serve como orientação básica para este critério de sucesso.

Diretriz 1.4 – discernível

Facilitar a audição e visualização de conteúdos aos usuários. Tornar a experiência do usuário mais simples e de fácil compreensão é a melhor forma de garantir que a informação da página seja transmitida de forma adequada.

1.4.1 Utilização da Cor

A documentação do WCAG é enfática nesse sentido: Nunca utilize somente a cor para transmitir informação ao seu usuário. Pessoas com algum tipo de daltonismo e usuários que não enxergam podem ter dificuldades em utilizar um website que oriente suas ações somente por cores. Por exemplo, em uma tabela que informa que o status verde significa "em andamento" e vermelho "parado". Se utilizarmos somente a cor para informar o status, pessoas com algum tipo de daltonismo podem não compreender a tabela.

Com relação ao contraste adequado entre o conteúdo e o fundo, o W3C orienta que ele seja de no mínimo 4,5:1 exceto para textos ampliados (que devem ter um contraste mínimo de 3:1), textos decorativos e logotipos que não têm requisito de contraste.

1.4.2 Controle de áudio

Se um som em uma página Web começar a tocar automaticamente (áudio ou vídeo) durante mais de 3 segundos, deve estar disponível um mecanismo para fazer uma pausa ou parar o som e vídeo, ou deve disponibilizar um mecanismo para controlar o volume do som, independentemente de todo o nível de volume do sistema.

6.4 Princípio 2: operável

Os componentes de interface de usuário e a navegação têm de ser operáveis.

O que isso significa? Que os usuários devem ser capazes de operar a interface; a interface de interação não pode exigir interação que o usuário não possa executar.

Diretriz 2.1 – acessível via teclado

Toda a funcionalidade da sua página deve ser acessível por teclado, garantindo que não existam barreiras de acesso a usuários que não possam utilizar um mouse, por exemplo.

2.1.1 Teclado

Este critério de sucesso define que toda a funcionalidade deve estar acessível via teclado. Para que esse critério seja cumprido, existem algumas técnicas que auxiliam o desenvolvimento.

Além de utilizar os links (âncoras) do HTML, que, por padrão, já garantem o foco via teclado, utilizar funções de foco em conjunto com funções de passar o mouse ajuda a garantir que o foco via teclado seja feito. No caso de uso de JavaScript, o exemplo a seguir é válido:

```
<a href="menu.php"
    onmouseover="swapImageOn('menu')"
    onfocus="swapImageOn('menu')"
    onmouseout="swapImageOff('menu')"
    onblur="swapImageOff('menu')">

    <img id="menu" src="menu_off.gif" alt="Menu">
</a>
```

Utilize o evento `onfocus` em conjunto com `onmouseover`. O mesmo se aplica para funções de "tirar o mouse".

E, em CSS, a regra é a mesma.

```
a:hover,
a:focus {
    background-color:#FFF;
}
```

2.1.2 Sem bloqueio do teclado

Da mesma forma que ter o conteúdo acessível via teclado é importante, o desenvolvedor deve evitar barreiras de teclado para o usuário. Durante a navegação via teclado, o usuário não deve ficar preso em um determinado ponto da página. Faça testes e navegue utilizando somente o teclado para identificar possíveis barrei-

ras (como scripts) que não permitam a navegação e operação correta da página ou aplicação.

Diretriz 2.2 – tempo suficiente

Deixar tempo suficiente para que o usuário possa cumprir as tarefas da página, como ler e utilizar o conteúdo. Atualizações automáticas e atividades que não permitem que o usuário possa parar ou pausar uma ação dificultam o acesso ao conteúdo.

2.2.1 Ajustável por temporização

Proporcione a possibilidade de desligar, ajustar ou pausar o tempo para que o usuário possa executar sua ação em tempo suficiente. As exceções são para ações em tempo real e quando o limite de tempo é essencial, como um leilão online.

2.2.2 Colocar em pausa, parar, ocultar

O usuário deve ser capaz de pausar, parar ou ocultar quaisquer informações em movimento, em modo intermitente ou em deslocamento, que sejam iniciadas automaticamente, durem mais de cinco segundos e sejam apresentadas em paralelo com outro conteúdo.

Diretriz 2.3 – convulsões

Não crie conteúdos que possam causar convulsões para os usuários.

2.3.1 Três flashes ou abaixo do limite

Evitar publicar conteúdo que exiba três flashes no período de um segundo. Pessoas com fotosensibilidade podem ter convulsões, dependendo da forma como o conteúdo pisque em uma tela. A documentação do W3C fornece mais detalhes sobre o tamanho de uma área segura para o piscar de flashes de luz que não sejam em toda a tela.

Diretriz 2.4 – navegável

Fornecer formas de ajudar os usuários a navegar, localizar conteúdos e determinar o local em que eles estão em uma página web. O usuário deve ter controle sobre o conteúdo e sobre sua localização na página Web, o que deve facilitar a navegação e busca por informação.

2.4.1 Ignorar blocos

Possibilite que o usuário possa saltar ou ignorar áreas específicas da página. As técnicas para cumprir esse critério são as seguintes:

Adicione um link no topo da página para o conteúdo principal. Essa técnica beneficia não só pessoas que utilizam softwares leitores de tela como também pessoas com mobilidade reduzida, já que possibilita o acesso via teclado ao conteúdo principal sem a necessidade de navegar por todos os links da página.

```
<a href="#skip">Saltar conteúdo</a>
.....
<h2 id="skip">Conteúdo principal</h2>
```

Essa técnica pode ser utilizada para saltar não só para o conteúdo principal como também saltar apenas blocos específicos de conteúdo.

Utilize os cabeçalhos para separar as seções do documento.

A estrutura de cabeçalhos é importantíssima, pois além de garantir a semântica de cada bloco, definindo seu grau com relação ao cabeçalho anterior, possibilita que usuários de tecnologias assistivas utilizem esse recurso para navegar apenas pelos cabeçalhos. Por exemplo, usuários de softwares leitores de tela navegam pelos cabeçalhos pressionando a tecla "H" do teclado, mas também podem navegar pelos números de 1 a 6 do teclado numérico para navegar pelo nível de cabeçalho desejado (H1-H6).

2.4.2 Página com título

O título deve descrever claramente o tópico ou a finalidade da página. Deve ficar claro ao usuário o que ele vai encontrar ao acessar aquela página web. O elemento `<title>` é um dos primeiros elementos a ser lido por um software leitor de tela.

```
<title>The World Wide Web Consortium</title>
```

2.4.3 Ordem do foco

A ordem de navegação da página deve ser sequencial naturalmente. Caso isso não seja possível, devem-se proporcionar formas para garantir essa navegação sequencial sem que o significado e a operabilidade sejam perdidos. As técnicas para garantir esse critério são simples.

Crie uma sequência lógica de tabulação utilizando os recursos do HTML.

Elementos de link e formulários já proporcionam por padrão o foco por teclado. Por esse motivo, utilizar recursos de forma linear e sequencial é o recomendado. Caso não seja possível, pode-se fazer uso do atributo `tabindex` para obter o resultado adequado na navegação.

```
<a href="index.html" tabindex="1">Link para a página principal</a>
<input type="text" name="nome" tabindex="2">
```

2.4.4 Finalidade do link (em contexto)

Garantir que a finalidade de cada link possa ser determinada pelo texto dentro do próprio link. Evitar o uso de "clique aqui" ou "veja mais" são alguns dos exemplos. O usuário deve ser capaz de compreender aquele link fora de contexto. Por exemplo:

```
<a href="rotas.html">
  Rotas para chegar ao consultório
</a>

<a href="rotas.html">
   <img src="rota.gif" alt=" Rotas para chegar ao consultório ">
</a>
```

6.5 Princípio 3: compreensível

A informação e a operação da interface de usuário têm de ser compreensíveis.

O que isso significa? Que os usuários devem ser capazes de compreender as informações, bem como o funcionamento da interface do usuário; o conteúdo ou operação não pode ir além de sua compreensão.

Diretriz 3.1 – legível

Tornar o conteúdo web compreensível e legível para o seu usuário. Isso está relacionado a interferências além do código, como um texto bem escrito e definido, mas alguns recursos que melhoram a acessibilidade podem ser implementados de forma que auxiliem a leitura e compreensão do texto.

3.1.1 Linguagem da página

Definir o idioma da página é fundamental para que ferramentas de busca, navegadores e tecnologias assistivas consigam interpretar o conteúdo em seu idioma

correto. Utilize o atributo `lang` no elemento `<html>` para declarar o idioma da página no início do seu código.

```
<html lang="pt-br">
```

Você pode mudar o idioma a qualquer momento na página, utilizando os elementos adequados.

```
Este texto está em português <span lang="en"> and in english</span>
```

Diretriz 3.2 – previsível

Faça com que as páginas Web surjam e funcionem de forma previsível. Surpreender o usuário em tarefas rotineiras ou na busca por informação pode atrapalhar a efetividade da sua página Web.

3.2.1 Em foco

Um componente da página ou aplicação não inicia uma alteração de contexto quando recebe o foco. Um menu dropdown que inicia uma ação apenas ao fazer foco ou um link que executa uma função automaticamente ao receber foco pode dificultar o acesso de pessoas que navegam por teclado, como usuários de softwares leitores de tela.

3.2.2 Em Entrada

Alterar a definição de um componente de interface de usuário não provoca, automaticamente, uma alteração de contexto, a menos que o usuário tenha sido avisado sobre essa situação antes de utilizar o componente. Sempre que algo inesperado em uma página Web estiver programado para acontecer, o usuário deve ser informado desse tipo de alteração.

Diretriz 3.3 – assistência de entrada

Esta diretriz orienta os desenvolvedores a ajudarem seus usuários a evitar e corrigir erros. Informações sobre como preencher um campo de formulário ou sobre como o usuário deve utilizar o sistema/aplicação são muito úteis não só para pessoas com deficiência, mas também para pessoas que não estão familiarizadas com o sistema.

3.3.1 Identificação do erro

Sempre descrever ao usuário os tipos de erros encontrados ao executar uma ação. Quando um usuário preenche um campo de formulário de forma inadequada, o sistema deve informá-lo onde estão esses erros e como solucioná-los.

3.3.2 Etiquetas ou instruções

Além de identificar adequadamente os campos de formulário utilizando o elemento `label` (ver diretriz 1.1, critério de sucesso 1.1.1, situação C), os formulários devem conter informações relevantes sobre como o usuário deve preenchê-lo. Para evitar os erros mais comuns, como formas de preencher determinados campos ou quais são obrigatórios, é importante orientar o usuário sobre como preencher corretamente o formulário.

6.6 Princípio 4: robusto

O conteúdo tem de ser robusto o suficiente para poder ser interpretado de forma concisa por diversos agentes do usuário, incluindo tecnologias assistivas.

O que isso significa? Que os usuários devem ser capazes de acessar o conteúdo, especialmente conforme as tecnologias evoluem; como a tecnologia e os agentes de usuário evoluem, o conteúdo deve permanecer acessível.

Diretriz 4.1 – compatível

Essa diretriz orienta a maximizar a compatibilidade com atuais e futuros agentes de usuário, incluindo tecnologias assistivas.

4.1.1 Análise

Avalie se o conteúdo está estruturado de forma adequada: se os elementos dispõem de marcas de início e fim completas, se estão encaixados conforme a especificação, se não têm atributos duplicados e se todos os Ids são únicos e exclusivos.

Uma boa forma de verificar esse critério de sucesso é fazer a validação do MarkUp pelas ferramentas do W3C. Ferramentas de validação de HTML (http://validator.w3.org/) e CSS são muito úteis para identificar possíveis problemas na marcação do código, que podem comprometer a acessibilidade da sua página. Ferramentas que fazem verificação automática de contraste e de acessibilidade também são úteis para auxiliar a encontrar erros que poderiam ser comprometedores.

A WAI disponibiliza uma lista com diversas ferramentas que ajudam a fazer uma verificação automática de acessibilidade. A WAI não se responsabiliza nem verifica ou endossa nenhuma desses validadores. Seu *disclaimer* é claro: nenhuma ferramenta automática pode determinar a acessibilidade de um website. A lista com as ferramentas está disponível em http://www.w3.org/WAI/ER/tools/.

6.7 Conclusão

As diretrizes de acessibilidade para conteúdo Web auxiliam o desenvolvedor a tornar seu código acessível para pessoas com deficiência. Seguir as orientações de acessibilidade do W3C garante não só um código acessível, mas também uma página mais leve e que vai ter performance melhor em navegadores e sistemas de busca.

Este artigo mostra de forma muito simples que boa parte das principais questões relacionadas à acessibilidade na Web estão diretamente ligadas com o uso correto dos padrões do W3C. Utilizar os padrões de forma adequada e validar seu código é um ótimo primeiro passo para colocar a acessibilidade na Web na rotina do desenvolvimento.

Acessibilidade na Web é uma questão importante não somente para beneficiar pessoas com deficiências. Nós, em algum momento da nossa vida vamos precisar da acessibilidade, ou porque estamos envelhecendo e perdendo a acuidade visual, auditiva e destreza motora ou porque podemos estar com uma das mãos ocupadas ao acessar a Web de um telefone celular. Acessibilidade na Web é um dos pilares do W3C e vai de encontro com seu maior princípio. Uma Web para todos.

Sobre o autor

Reinaldo Ferraz é especialista em desenvolvimento Web do W3C Brasil. Formado em desenho e computação gráfica e pós-graduado em design de hipermídia pela Universidade Anhembi Morumbi em São Paulo. Trabalha há mais de 12 anos com desenvolvimento Web. Coordenador do Prêmio Nacional de Acessibilidade na Web e do Grupo de Trabalho em Acessibilidade na Web e representante do W3C Brasil em plenárias técnicas do W3C.

Capítulo 7

Aplicações web super acessíveis com WAI-ARIA

Acessibilidade é um tema importantíssimo. O capítulo anterior, do Reinaldo Ferraz, tratou da base de acessibilidade na Web e as diretrizes do WCAG. Mas há muito mais a ser visto. Em especial, o **WAI-ARIA**, foco deste capítulo, que nos vai permitir refinar ainda mais os aspectos de acessibilidade com novas semânticas e foco em aplicações mais complexas.

WAI-ARIA define uma forma de tornar o conteúdo e aplicativos Web mais acessíveis a pessoas com deficiência. Ele contribui especialmente com conteúdo dinâmico e interface de controles de usuário avançadas desenvolvidos com Ajax, HTML, JavaScript e tecnologias relacionadas. Além disso, WAI-ARIA é um padrão e recomendação do W3C.

Entenda o significado — WAI (*Web Accessibility Initiative*), ARIA (*Accessible Rich Internet Applications*).

Em resumo, ARIAs servem pra dar **semântica** para os leitores de tela, que inter-

pretarão com mais significado aquele elemento do seu HTML, além de estender esse significado das interações com o projeto Web. Vamos ver exemplos práticos adiante.

Importante ressaltar que as ARIAs se aplicam em qualquer versão do HTML, ou seja, não dependem do HTML5 para se ter um projeto Web acessível.

Dá pra se afirmar que a acessibilidade está presente em todos os passos de desenvolvimento de um projeto Web, e ela é muito importante, já que está ligada diretamente ao usuário, que precisa entender, compreender e interagir com seu projeto.

Existem ferramentas automatizadas (http://www.w3.org/WAI/ER/tools/) que ajudam na avaliação de acessibilidade. Contudo, nenhuma ferramenta sozinha é capaz de determinar se um site cumpre todos esses itens. Uma avaliação feita por humanos é essencial para determinar se um projeto Web está acessível.

7.1 Leitores de tela

Existem vários leitores de tela no mercado, no entanto, os mais utilizados são: **JAWS** (Windows / pago), **NVDA** (Windows / gratuito) e **Voice Over** (nativo no Mac). São eles os responsáveis por ajudar deficientes visuais a navegarem na internet e a consumirem todo seu conteúdo. Por isso, é fundamental pensarmos na acessibilidade de uma maneira completa.

Quando falamos de implementar acessibilidade em projetos Web, é importante citar que esses testes começam utilizando apenas o teclado e, se todas as principais funcionalidades ocorrerem normalmente, passa-se para o próximo passo que é testar com leitores de tela e, consequentemente, as implementações das ARIAs.

WAI-ARIA pode ter 2 tipos de implementação. Utilizando as **roles** ou **estados e propriedades**. Eles são **atributos simples** adicionados em elementos do HTML que auxiliam nessa interação com os leitores de tela.

Barra de Progresso

Vamos ver um exemplo simples, que implementa uma barra de progresso, um controle bastante comum de aparecer em interfaces ricas. O HTML5 até possui um componente pronto hoje com `<progress>`, mas ele não tem suporte em navegadores antigos e pode ser um pouco complicado de estilizar. Por isso, muitas bibliotecas de componentes recriam essa funcionalidade com elementos comuns como `div` e usando JavaScript.

O problema de usar `div` simples para isso é que o conteúdo não é acessível. O leitor de tela não sabe que aquilo é uma barra de progresso. Usando WAI-ARIA,

podemos adicionar essa semântica:

```
<div class="my-progress-bar" role="progressbar"
     aria-valuenow="75" aria-valuemin="0"
     aria-valuemax="100" style="width: 75%;">

  75%

</div>
```

Repare no uso do atributo `role="progressbar"`. Ele indica semanticamente que o `div` representa uma barra de progressos. Além disso, usamos outras propriedades *aria-* para indicar valores semânticos que antes estavam apenas perdidos no texto ou no design. Em especial, `aria-valuemax` representa o valor máximo da barra de progresso, assim como `aria-valuemin` representa o mínimo. E a propriedade `aria-valuenow` representa o valor atual.

Todas essas informações são essenciais para boa acessibilidade, além, claro, dos aspectos básicos que vimos no capítulo anterior, como sempre prover acesso via teclado.

7.2 Roles

O atributo `role` no HTML redefine o significado semântico de um elemento, melhorando sua interpretação pelos leitores de tela. As roles são classificadas da seguinte maneira:

1) *Abstract Roles* — definem conceitos gerais, mas não servem para marcar conteúdo;

2) *Widget Roles* — como o próprio nome diz, servem para definir widgets de interface do usuário;

3) *Document Structure Roles* — usados em estruturas que organizam o conteúdo em uma página, que geralmente não é interativo;

4) *Landmark Roles* — regiões importantes de navegação.

Veja outro exemplo de implementação de um controle de Abas usando `role="tab"` para representar a aba em si e `role="tabpanel` que representa o conteúdo mostrado naquela aba:

```
<ul class="tab-panel">
   <li>
      <a href="#panel1" id="tab1" role="tab"
           class="active" aria-selected="true">
          Aba 1
      </a>
   </li>
   <li>
      <a href="#panel2" id="tab2" role="tab">Aba 2</a>
   </li>
   <li>
      <a href="#panel3" id="tab3" role="tab">Aba 3</a>
   </li>
</ul>

<div id="panel1" role="tabpanel" aria-labelledby="tab1">
  Primeira aba
</div>
<div id="panel2" role="tabpanel" aria-labelledby="tab2">
  Segunda aba
</div>
<div id="panel3" role="tabpanel" aria-labelledby="tab3">
  Terceira aba
</div>
```

Repare ainda que usamos mais duas propriedades *aria-*. A `aria-labelledby` amarra o elemento do conteúdo das abas com o elemento que o descreve. Na prática, aponta para o `id` do elemento que representa a aba.

Já a propriedade `aria-selected` permite indicarmos qual aba está selecionada atualmente. Se implementássemos um controle via JavaScript para troca de abas, por exemplo, basta alterar o valor desse atributo:

```
$('[role=tab]').click(function(){
    $('[role=tab][aria-selected=true]').attr('aria-selected', 'false');
    $(this).attr('aria-selected', 'true');
});
```

Implementação em um Menu

Menus são links comuns que montamos no HTML. Para os "videntes", é bem óbvio saber que ali existe um menu, mas e para um deficiente visual? Pode-

mos melhorar isso, dar mais significado implementando as arias como atributo `role="menuitem"`. Além disso, a `role="navigation"` serve para que os landmarks dos leitores possam entender que ali é uma navegação.

No caso de existir submenus, implemente `aria-expanded="false"` e `aria-hidden="true"` e altere os valores (via JavaScript) para `aria-expanded="true"` `aria-hidden="false"` conforme o usuário expande os submenus. Com isso, o leitor entenderá que essa lista de links está em outro nível. Importante: submenus devem ser acessíveis via teclado e não somente com o mouse.

```
<ul role="navigation">
    <li><a href="#" role="menuitem">menu 1</a></li>
    <li>
    <a href="#" role="menuitem">menu 2</a>
        <ul aria-expanded="false" aria-hidden="true">
        <li><a href="#" role="menuitem">Submenu 1</a></li>
        <li><a href="#" role="menuitem">Submenu 2</a></li>
        </ul>
    </li>
</ul>
```

Implementação em um Modal

Modal é aquela famosa caixa, conhecida como *lightbox* (devido ao plugin), que trazem informações com um tipo de janela na própria página. Utilizando as Arias podemos deixá-las mais sensíveis aos leitores de tela.

O atributo `role="dialog"` informará ao leitor que ele está dentro de um caixa de diálogo.

Além disso, o `aria-labelledby` possui valor do ID do elemento que descreve a caixa de diálogo, que pode ser o título, assim o leitor informa o usuário o nome desse modal.

```
<div class="modal" role="dialog" aria-labelledby="titulo">
    <div class="modal-header">
        <button type="button" aria-label="Fechar Modal" class="close">
            ×
        </button>

        <h3 id="titulo">Título da Modal</h3>
    </div>
```

```
    <div class="modal-body">
        <p>Conteúdo do modal de exemplo</p>
    </div>
    <div class="modal-footer">
        <a href="#" class="btn">Salvar</a>
        <a href="#" class="btn">Fechar</a>
    </div>
</div>
```

Outro atributo muito utilizado é `aria-label` aplicado no botão de *fechar*. O botão foi construído com apenas um *"x"*, que é adequado para usuários visuais saberem que serve para fechar. Mas, para leitores de tela, usamos uma frase melhor, com `aria-label="Fechar"`. O leitor vai interpretar esse valor na hora de ler para o usuário.

7.3 Formulários acessíveis

Os formulários são as formas mais comuns para a prestação de serviços interativos, tendo a finalidade de coletar dados e informações a serem processados. Com relação à acessibilidade, eles podem vir a ser um obstáculo, caso seus desenvolvedores não possuam conhecimentos sobre o assunto. É preciso garantir que o formulário seja fácil de navegar e de entender, ou seja, o usuário deverá ter acesso e controle sobre todos os elementos do formulário e deverá conseguir compreender as informações contidas nele.

No capítulo anterior do livro, o Reinaldo mostrou várias técnicas para se trabalhar com formulários de forma acessível. O principal é uso de `<label>` e do atributo `title` corretamente. Mas é possível ainda controlar a ordem lógica de navegação com `tabindex`.

Usamos o `role="form"` no formulário e alguns atributos *aria-* para melhorar a acessibilidade. Em especial, utilizando `aria-required="true"`, informaremos ao leitor de tela que o campo é obrigatório:

```
<form action="" method="post" role="form">
<fieldset>
    <legend>Título do Formulário</legend>
    <p>(*) campos obrigatórios</p>
    <div class="form-group">
        <label for="name">Nome</label>*
        <input class="form-control" name="name"
```

```
                aria-required="true" type="text" id="name">
      </div>
      <div class="form-group">
          <label for="email">E-mail</label>*
          <input class="form-control email" name="email"
                aria-required="true" type="text" id="email">
      </div>
      <div class="form-group">
          <label for="msg">Mensagem</label>*
          <textarea class="form-control" cols="30" rows="10"
                    name="msg" aria-required="true" id="msg">
          </textarea>
      </div>

      <input class="btn btn-primary" type="submit"
            aria-label="Enviar dados do formulário simples"
            value="Enviar dados">
</fieldset>
</form>
```

Muitas páginas têm problemas com acessibilidade por se preocuparem apenas em expressar visualmente as restrições. Por exemplo, um formulário que indica os campos obrigatórios apenas através da mudança de cor do input. Um usuário que não pode ver, não saberá quais campos são de preenchimento obrigatório. A solução é justamente usar o `aria-required`.

Repare que usamos novamente no botão um `aria-label` para sobrescrever a descrição utilizada pelo leitor de tela para o componente.

Mensagem de erros em formulários

Quando um erro de entrada de dados for automaticamente detectado, seja pelo não preenchimento de campos obrigatórios ou pelo preenchimento incorreto de algum campo, o item que apresenta erro deve ser identificado e o usuário deve ser informado por meio de texto.

Após o envio das informações, caso exista algum campo incorreto, o mesmo deve receber o foco, e nele deverá conter uma mensagem informando claramente ao usuário o real motivo do problema, para que possa ser resolvido, permitindo o envio dos dados.

Para melhorar isso aos usuários com deficiência visual, podemos utilizar WAI-ARIA, implementando `aria-invalid="true"`, que informa ao leitor que existe

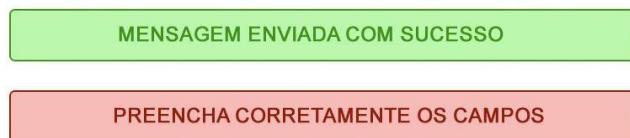
erro no campo. E usar `aria-label` para descrever exatamente o erro ocorrido.

```
<form action="" method="post" role="form">
<fieldset>
    <legend class="no-border">Formulário Simples</legend>
    <p>(*) campos obrigatórios</p>

    <div class="form-group has-error">
        <label for="name">Nome</label>*
        <input class="form-control required" name="name"
               aria-required="true" type="text" id="name"
               aria-invalid="true"
               aria-label="Não pode ficar em branco">
        <p for="name" class="help-block">Não pode ficar em branco</p>
    </div>

    <div class="form-group has-error">
        <label for="email">E-mail</label>*
        <input class="form-control email required" name="email"
               aria-required="true" type="text" id="email"
               aria-invalid="true"
               aria-label="Preencha com um e-mail válido">

        <p for="email" class="help-block">
            Preencha com um e-mail válido
        </p>
    </div>

    <div class="form-group has-error">
        <label for="msg">Mensagem</label>*
        <textarea class="form-control required" cols="30"
                  rows="10" name="msg" aria-required="true"
                  id="msg" aria-invalid="true"
                  aria-label="Não pode ficar em branco">
        </textarea>
        <p for="msg" class="help-block">Não pode ficar em branco</p>
    </div>

    <input class="btn btn-primary" type="submit"
           aria-label="Enviar dados do formulário simples"
           value="Enviar dados">
```

```
</fieldset>
</form>
```

*Nome:

O campo Nome é obrigatório

*Email:

O campo Email é obrigatório

aria-required="true" aria-invalid="true" aria-label="O campo Nome é obrigatório"

No exemplo, simulamos o envio do formulário e o servidor gerando o erro com os atributos ::aria** corretos. O mesmo poderia ter sido feito com JavaScript para uma validação dinâmica.

Alertas

Seguindo o fluxo após submeter seu formulário ao servidor, podemos a informação de sucesso ou de erro, conforme a imagem a seguir:

MENSAGEM ENVIADA COM SUCESSO

PREENCHA CORRETAMENTE OS CAMPOS

O atributo `role="alert"` facilita para o leitor de tela que esse elemento é um alerta, já que ele receberá o foco logo em seguida.

```
<div class="alerta" role="alert">
    <p>Mensagem enviada com sucesso</p>
</div>
```

Ao criar o código da figura anterior, será necessário a implementação via JavaScript que o elemento tenha foco e receba o atributo `tabindex="-1"`, assim o leitor de tela consegue dar prioridade e mostrar ao usuário a informação que ali consta.

7.4 Role Document Structure

Alguns *roles* representam apenas estruturas que organizam o conteúdo na página. Por exemplo, um `article` para indicar um pedaço da página que tem um conteúdo independente.

O HTML5 inclusive possui uma tag `<article>` para esse propósito. Mas podemos adicionar a mesma semântica sem HTML5 usando WAI-ARIA:

```
<article role="article">
   <p>Conteúdo</p>
</article>

<div role="article">
   <p>Conteúdo</p>
</div>
```

Há outros roles simples com esse propósito, como `list`, `img`, `note`, `region` e outros. Consulte a lista: http://www.w3.org/TR/wai-aria/roles#document_structure_roles.

7.5 Landmarks

As *landmarks* são *roles* especiais com objetivo de melhorar a navegação do usuário na página. A ideia é que o usuário possa saltar entre *landmarks* com rapidez.

Os leitores de tela em sua maioria já interpretam corretamente, mas alguns ainda não.

1) Jaws V.11 e maior tem suporte completo

2) ChromeVox V.1 e maior tem suporte completo

3) VoiceOver V.3 e maiores apoios todos os pontos de referência, exceto "form"

4) NVDA 2 suporta todos as landmarks, com exceção do "application"

5) Window Eyes V.7 não suporta landmarks

Nós já usamos uma landmark antes, como `role="form"`. Existem várias outras.

Conheça agora para que cada LANDMARKs serve e implemente corretamente, lembrando que elas são regiões importantes de navegação dentro do seu projeto.

main

Conteúdo principal em um documento. Isto marca o conteúdo que está diretamente relacionado com o tema central da página.

```
<main role="main">
```

O interessante é que, por ser uma *landmark* navegável, o usuário pode saltar direto para o conteúdo principal, facilitando muito a usabilidade para leitores de tela. Em leitores compatíveis, faz o papel do link de "saltar" que vimos no capítulo anterior.

banner

A região que contém o título principal ou o título do site. A maior parte do conteúdo de um `banner` é orientada para o local, ao invés de ser específico da página. Conteúdo orientado para o site geralmente inclui coisas como o logotipo do patrocinador local, o principal título da página, e uma ferramenta de busca específica do site. Tipicamente este aparece no topo da página abrangendo toda a largura.

```
<div role="banner">
```

complementary

Qualquer seção do documento que suporta, mas é separável do conteúdo principal, mas é significativo por si só, mesmo quando separado dele.

```
<aside role="complementary">
```

contentinfo

No contexto da página, é onde se aplicam informações. Por exemplo, as notas de rodapé, direitos de autor, os links para declarações de privacidade etc.

```
<footer role="contentinfo">
```

form

A região que contém um conjunto de itens e objetos que, como um todo, se combinam para criar um formulário.

```
<form action="" method="post" role="form">
```

navigation

Uma coleção de links apropriados para o uso ao navegar no documento ou documentos relacionados.

```
<nav role="navigation">
    <a href="">Link 1</a>
    <a href="">Link 2</a>
    <a href="">Link 3</a>
</nav>
```

search

A ferramenta de pesquisa de um documento Web. Isto é tipicamente um formulário usado para enviar solicitações de pesquisa sobre o site ou a um serviço de busca mais geral.

```
<div role="search">
```

application

Uma região declarada como uma aplicação Web, ao contrário de um documento Web.

```
<div role="application">
```

Para conhecer outras *roles*, basta acessar:
http://www.w3.org/TR/wai-aria/roles#roles_categorization

7.6 Conclusão

Criar projetos Web acessíveis não é difícil, nem custoso e muito menos impacta no tempo de desenvolvimento, desde que seja implementado no seu início.

No Brasil, nenhum dos grandes e-commerces é acessível, impossibilitando muitos de comprarem e tornarem-se independente na web, ou seja, há uma grande fatia do mercado ainda a ser suprida. São clientes fiéis que estão deixando de ser atendidos.

Implementar WAI-ARIA é fácil, simples e, ao testarmos, funciona magicamente, basta ler e entender a documentação.

Implementar acessibilidade não é caridade, e sim, ter a consciência de que no país há milhões de pessoas com algum tipo de deficiência, sendo ela permanente

ou temporária, e dessa maneira estaremos possibilitando a todos eles o acesso a seu conteúdo.

O W3C Brasil incentiva essas práticas. Ele criou o Prêmio Nacional de Acesssibilidade na Web — basta acessar o link para saber mais detalhes: http://premio.w3c.br/

Sobre o autor

Deivid Marques é um simples cara do interior que acabou transformando um hobby em profissão, já que adora esse mundo da internet. É desenvolvedor Front-end com foco em usabilidade e há 5 anos mora em São Paulo. Além disso, é um dos organizadores do evento Front in Sampa e faz parte do Grupo de Trabalho de Acessibilidade do W3C Brasil.

Capítulo 8

APIs geniais da Web moderna

É incrível o que os browsers modernos conseguem fazer. Para quem acompanhou a Web alguns anos atrás, é quase difícil acreditar aonde chegamos. A revolução pós-HTML5 trouxe capacidades fantásticas para os navegadores. Neste capítulo, vamos falar de algumas dessas novidades.

8.1 WebStorage (localStorage e sessionStorage)

Aplicações baseadas em banco de dados, fizeram, e ainda fazem parte do cotidiano de todo desenvolvedor Web, Desktop, Mobile (plataformas nativas) etc. Armazenar informações é algo imprescindível na grande maioria dos softwares.

A plataforma Web por si só, até pouco tempo atrás, sem auxílio de serviços de backend, não era capaz de persistir informações de maneira estruturada e bem organizada no browser. Sim, algumas soluções alternativas sempre existiram, mas até hoje, não passam de soluções alternativas, que na grande maioria das vezes, não satisfazem a real necessidade de uma aplicação Web.

O que tínhamos disponível até agora quando queríamos guardar uma informação no navegador do cliente?

Nome da janela do Browser

Os browsers possuem uma propriedade no escopo principal, a propriedade `name`, do objeto `window`.

```
/* Acessando o valor da propriedade */
var data  = window.name;

/* Atribuindo valor */
window.name = 'foo';
```

Esta propriedade apenas atribui um nome à janela. O porém é que a string que pode ser armazenada nesta propriedade pode ter, em alguns casos, até 2MB.

Outro porém é que esta propriedade guarda o seu estado, ou seja, mesmo que a página seja atualizada, o valor continua disponível.

Por muitos anos, alguns desenvolvedores utilizaram esta técnica de gosto duvidoso como uma solução para armazenamento de dados.

Atenção: só estou mencionando esta técnica aqui para fins de curiosidade, não a utilize em suas aplicações. Como veremos em seguida, há melhores maneiras de se armazenar dados no browser.

Cookies

Ah, os Cookies.

A primeira especificação foi feita pelo Lou Montulli, um funcionário da Netscape na época (meados de 1997). Lou estava tentando resolver um problema relacionado ao primeiro carrinho de compras da Web. A sua especificação original continha as informações básicas de como os "Cookies" deveriam funcionar e foi formalizada na RFC 2109, que é a referência para a grande maioria dos browsers.

> **Mais da história dos Cookies**
>
> Recomendamos alguns links para você conhecer o surgimento dos cookies:
>
> http://www.nczonline.net/blog/2009/05/05/http-cookies-explained/
> http://curl.haxx.se/rfc/cookie_spec.html
> http://tools.ietf.org/html/rfc2109

Mas o que é um Cookie, afinal?

Um Cookie é basicamente um pequeno arquivo texto que é armazenado na máquina do usuário pelo browser. Este arquivo de texto é enviado para o servidor a cada request, para que ele possa processar algum tipo de informação relevante. Cookies podem ser criados tanto pelo servidor (aplicação backend), quanto pelo cliente (aplicação frontend).

Mas por que falar tanto sobre Cookies?

Bom, Cookies ainda são extremamente úteis atualmente para diversas situações, inclusive, armazenar dados. Mas o objetivo do Cookie não é apenas este, e por isso, ele acaba sendo ineficaz em tarefas básicas.

É aí que surge a **API WebStorage**, uma das novidades dos browsers modernos.

O que temos de novo?

localStorage

`localStorage` é um objeto JavaScript global, que nos fornece alguns métodos para armazenamento de dados no browser.

Sua API é bem simples e baseada em chave-valor. Vamos a um exemplo, no qual queremos gravar o *nome* e a *idade* de um usuário:

```
window.localStorage.setItem('nome', 'John Smith da Silva');
window.localStorage.setItem('idade', 29);
```

Note que utilizamos o métdodo `setItem` para armazenar um valor, passando uma chave como referência. Para resgatar o valor previamente armazenado, utilizamos o método `getItem`, exemplificado a seguir:

```
var nome = window.localStorage.getItem('nome');
var idade = window.localStorage.getItem('idade');
```

```
console.log(nome); // John Smith da Silva
console.log(idade); // 29
```

Um detalhe importante: a API LocalStorage funciona apenas com String. Para passar um objeto, é preciso transformá-lo em uma String antes. Veja o exemplo a seguir:

```
window.localStorage.setItem('usuario',
        { nome: "John Smith da Silva", idade: 29 });
```

Neste exemplo, temos um falso-positivo, pois a chamada ao método `getItem` aparentemente funciona. De fato, algum valor foi atribuído para a chave `usuario`, porém, não o valor desejado. A chave `usuario` armazenou o valor `"[object Object]"`, que é o mesmo que o resultado do uso do método `toString` no objeto. Ou seja, ele transformou o objeto em String, mas não do jeito que gostaríamos.

A maneira mais recomendada para trabalhar com objetos em `localStorage`, é utilizar **JSON**. Exemplo:

```
var usuario = { nome: "John Smith da Silva", idade: 29 };
window.localStorage.setItem('usuario', JSON.stringify(usuario));
```

Para recuperar o valor como objeto, utilizamos o método `parse`, como no exemplo a seguir:

```
var usuario = JSON.parse(window.localStorage.getItem('usuario'));

console.log(usuario.nome); // John Smith da Silva
console.log(usuario.idade); // 29
```

Todas as informações salvas no `localStorage` ficam persistidas no navegador do usuário para sempre. Se ele fechar e abrir o navegador, as informações continuarão lá. As únicas maneiras de remover as informações é se o usuário limpar os dados do navegador ou se, em nosso código, chamarmos o método `removeItem`.

sessionStorage

`sessionStorage` possui implementação praticamente igual ao `localStorage`, com uma simples, mas importante diferença: a sessão.

O dados armazenados em `sessionStorage` só persistem para a sessão do usuário. Isto na prática quer dizer que o dado não mais existirá após o encerramento

da sessão, que é quando o usuário fechar o browser. Por este seu comportamento, `sessionStorage` é comumente utilizada para armazenar dados temporários.

```
window.sessionStorage.setItem('temp_user_data', 'efe2f3f201010101');
var temp_user_data = window.sessionStorage.getItem('temp_user_data');
console.log(temp_user_data); // efe2f3f201010101
```

Pontos importantes

Tenha em mente que as APIs de `localStorage` e `sessionStorage` são síncronas. Isto quer dizer que operações de resgate ou armazenamento de valores podem demorar e bloquear a execução de tarefas que deveriam ser executadas na sequência.

Para o uso moderado, ambas podem ser utilizadas sem problema, mas para uma grande quantidade de dados, pense em outras alternativas como o *IndexedDB* ou *WebSQL*.

Outro ponto que deve ser analisado com cuidado é a regra de *same-origin policy* (https://developer.mozilla.org/docs/Web/JavaScript/Same_origin_policy_for_JavaScript) .

Ambas APIs possuem restrição *cross-domain*. Isto quer dizer que:

- Dados armazenados em *http://www.dominio.com* não podem ser acessados de *http://www.foobar.com*;
- Dados armazenados em *https://www.dominio.com* não podem ser acessados de *http://www.dominio.com*;
- Dados armazenados em *http://www.dominio.com:8081* não podem ser acessados de *http://www.dominio.com:8080*;
- Dados armazenados em *http://foo.bar.com* não podem ser acessados de *http://bazinga.bar.com*.

Casos de uso

WebStorage pode ser utilizado em diversas situações, dependendo da aplicação e do seu julgamento da necessidade.

Um caso comum de uso é para melhorar a experiência em formulários.

Imagine um grande formulário, em que o usuário precisa preencher uma série de campos. Se, por algum motivo antes da submissão do formulário ao servidor,

acontecer algum problema, como crash no browser, queda de luz, falha de rede etc., os dados preenchidos pelo usuário serão perdidos. Se colocarmos a API WebStorage para funcionar aqui, precisamente falando do objeto `localStorage`, podemos ser proativos, e guardar os dados já preenchidos no browser. Basta uma implementação simples como a seguir:

```
var input = document.querySelector('#nome');
input.addEventListener('keyup', function (event) {
    var nome = document.querySelector('#nome').value;
    window.localStorage.set('campoNome', nome);
});
```

No exemplo anterior, no evento `keyup`, ou seja, após uma tecla ser pressionada, o valor preenchido é salvo em `localStorage`. Desta maneira, mesmo com algum imprevisto no meio do caminho, o dado estará salvo. Aí basta restaurar o valor quando o usuário voltar ao formulário.

Este é um exemplo simples, use sua criatividade e o poder da API WebStorage para criar soluções incríveis em suas aplicações.

Considerações finais

As APIs de `localStorage` e `sessionStorage` são simples, e por este fato foram facilmente adotadas pelos desenvolvedores.

Ao utilizar, entenda bem os seus benefícios e pontos negativos, para que sua aplicação funcione da maneira esperada. Para saber mais, consulte a especificação oficial e outras referências:

http://www.w3.org/TR/webstorage/

https://developer.mozilla.org/docs/Web/Guide/API/DOM/Storage

http://www.nczonline.net/blog/2012/03/07/in-defense-of-localstorage/

Atualmente, todos os browsers suportam WebStorage.

Web Storage - name/value pairs - Recommendation

Method of storing data locally like cookies, but for larger amounts of data (sessionStorage and localStorage, used to fall under HTML5).

Usage stats:	Global
Support:	88.34%
Partial support:	0.11%
Total:	88.45%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
								2.1		
								2.2		
						3.2		2.3		
						4.0-4.1		3.0		
	8.0			5.1		4.2-4.3		4.0		
	9.0			6.0		5.0-5.1		4.1		
	10.0	26.0	31.0	6.1		6.0-6.1		4.2-4.3	7.0	
Current	11.0	27.0	32.0	7.0	19.0	7.0	5.0-7.0	4.4	10.0	10.0
Near future		28.0	33.0		20.0					
Farther future		29.0	34.0		21.0					
3 versions ahead		30.0	35.0							

Figura 8.1: Compatibilidade dos browsers com WebStorage no início de 2014.

8.2 postMessage

Problemas de cross-domain

Todo desenvolvedor Web, especialmente na área de front-end, em alguma momento já se deparou com a necessidade de desenvolver algum tipo de comunicação *Cross-Domain*.

Cross-Domain é o nome popular para regra de *same-origin policy*, citada anteriormente na seção sobre Cookies. Recaptulando: por comunicação *cross-domain*, entenda o cenário onde um determinado dado precisa ser transferido para outro contexto em que o domínio, subdomínio, porta ou protocolo sejam diferentes.

Mas por que gostaríamos de fazer comunicação *Cross-Domain* na prática? Muitos sites hoje incluem widgets, scripts e recursos de outros domínios. É o botão de like do Facebook, um Google Maps embutido, uma propaganda de um servidor externo entre muitos outros casos. Ou ainda, em aplicações complexas, podemos optar por quebrar nosso código em várias pequenas aplicações que são chamadas todas na mesma página, geralmente com iframes.

Em todos esses casos, temos requisições de domínios diferentes na mesma página. E, além de incluir recursos de domínios diferentes, podemos querer conversar com eles. Como trocar o ponto em um mapa do Google Maps que foi incluído via iframe; ou ser avisado pelo botão de like do Facebook de quando o usuário gostou ou não da página.

Por razões de segurança, esse tipo de comunicação entre domínios não é permi-

tida. É a *same-origin policy* que comentamos. Só posso interagir com recursos da mesma origem.

Para passar por cima dessa restrição, o uso de **iframes** em aplicações Web é muito comum. E em algumas situaçãoes, é a única e melhor solução.

Imagine um cenário onde uma página hospedada no domínio http://www.casadocodigo.com.br (página pai) possui um `iframe` com `src` apontando para http://pagamentocdc.com.br (página filho). Neste caso, nem a página pai, nem a página filho possuem permissão de uso das APIs disponíveis para acesso direto a outros documentos.

As APIs de acesso a outros documentos são: `iframe.contentWindow`, `window.parent`, `window.open` e `window.opener`.

Na figura a seguir, um exemplo em que a página tenta usar a API `iframe.contentWindow` em um documento que está em outro domínio, ou seja, fere a regra de *same-origin policy*.

Figura 8.2: Cross origin iframe

A postMessage API

A API `postMessage` permite que dados sejam trafegados de maneira segura e controlada mesmo em contextos onde a regra de *same-origin policy* estaria sendo violada. Na prática, `postMessage` nos permite comunicação *cross-origin* na Web (diferentes domínios, subdomínios, porta e protocolo).

A sintaxe da API se divide em enviar e receber mensagens:

```
/* Enviando mensagens via postMessage */
target.postMessage(mensagem, origem, [transfer]);
```

Onde `target` é a referência para a janela onde a mensagem deve ser enviada, como por exemplo o objeto retornado pelo método `window.open`, ou a propriedade `contentWindow` de um `iframe`.

O parâmetro `mensagem` é a mensagem propriamente dita, os dados a serem enviados para a outra janela. É possível enviar uma String, ou um objeto literal.

`origem` especifica a URL da janela que a mensagem está sendo enviada. É preciso muita atenção ao especificar a origem, pois este parâmetro é que define o nível de permissão de envio de mensagens. É essencial para segurança, para não expor dados a origens erradas.

O uso do valor * é permitido, mas é definitivamente desencorajado, pois cria um túnel aberto, sem restrições de recebimento de mensagens de qualquer aplicação.

Postada a mensagem, a outra janela/iframe pode recebê-la.

```
/* Recebendo mensagens via postMessage */
window.addEventListener("message", function(event) {
   if (event.origin !== 'http://casadocodigo.com.br') {
      /*
        Só permitimos mensagens do
        domínio http://casadocodigo.com.br
      */
      return;
   }
}, false);
```

Note que a API nos oferece apenas o evento `message`. Dados oriundos de qualquer aplicação, enviados via `postMessage`, serão entregues pelo evento `message`. Assegure-se de apenas ouvir o evento `message` quando necessário, e **sempre** faça a verificação da origem para evitar qualquer falha de segurança.

Para recuperar mensagens, utilizamos propriedade `data` do objeto `event`, exemplificado a seguir:

```
/* Recebendo mensagens via postMessage */
window.addEventListener("message", function(event) {
   if (event.origin !== 'http://casadocodigo.com.br') {
      /*
        Só permitimos mensagens do
        domínio http://casadocodigo.com.br
      */
      return;
```

```
    }

    // exibe a mensagem num alerta
    alert(event.data);

}, false);
```

Vamos a um exemplo prático. A editora Casa do Código usa um serviço externo de pagamento. Uma aplicação, que chamaremos de *EditoraApp*, rodando no domínio http://www.casadocodigo.com.br quer receber e enviar dados para outra aplicação, `PagamentoApp`, que está no domínio http://pagamentocdc.com.br.

Isto vai ser feito através de um `iframe` que aponta para `PagamentoApp` incorporado na `EditoraApp`.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Editora Casa do Código</title>
    </head>
<body>
    <iframe id="PagamentoApp" src="http://pagamentocdc.com.br"></iframe>
</body>
</html>
```

A comunicação entre as aplicações é necessária para que a editora avise o sistema de pagamentos sempre que o usuário escolher um produto para comprar. E para que o sistema de pagamento avise a editora quando o pagamento for bem-sucedido.

Para essa comunicação ser possível, a aplicação `EditoraApp` precisa ouvir mensagens enviadas via `postMessage` e fazer o filtro da origem para receber apenas dados da aplicação `PagamentoApp`.

```
/* Recebendo mensagens na editora */
window.addEventListener("message", function(event) {
   if (event.origin !== 'http://pagamentocdc.com.br') {
      /*
        Só permitimos mensagens do sistema de pagamentos!
      */
      console.log("Acesso não permitido");
      return;
```

```
    }
}, false);
```

Feito isso, a aplicação `EditoraApp` estará apta a receber mensagens apenas do domínio http://pagamentocdc.com.br.

Agora, vamos enviar mensagens para a aplicação `PagamentoApp` passando os dados do produto que o usuário quer comprar.

```
var pgto = document.querySelector('#PagamentoApp').contentWindow;
pgto.postMessage('Coletânea Front-end', 'http://pagamentocdc.com.br');
```

Pronto. A aplicação `EditoraApp` agora já está ouvindo mensagens e enviando mensagens para a aplicação `PagamentoApp` de qual livro o usuário quer comprar.

Mas o fluxo ainda não está completo. A aplicação `PagamentodApp` também precisa ouvir mensagens da `EditoraApp` e, claro, filtrar a origem para receber dados apenas do domínio http://www.casadocodigo.com.br.

```
window.addEventListener("message", function(event) {
    if (event.origin !== 'http://www.casadocodigo.com.br') {
        /*
          Só permitimos mensagens do
          domínio http://www.casadocodigo.com.br
        */
        console.log("Acesso não permitido.");
        return;
    }

    // processa a cobrança com base no produto desejado
    cobranca.comprar(event.data);

}, false);
```

Para enviar mensagens de volta com a confirmação do pagamento, a aplicação `PagamentoApp` precisa acessar o `parent`, que é a janela principal:

```
parent.postMessage('Pagamento enviado','http://www.casadocodigo.com.br');
```

Agora o fluxo está completo, ambas aplicações podem enviar e receber mensagens uma para a outra.

Atualmente, todos os principais browsers suportam a API de `postMessage`.

Cross-document messaging - Working Draft

Method of sending information from a page on one domain to a page on a different one (using postMessage)

Usage stats:	Global
Support:	78.91%
Partial support:	12.89%
Total:	91.8%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
								2.1		
								2.2		
						3.2		2.3		
						4.0-4.1		3.0		
	8.0			5.1		4.2-4.3		4.0		
	9.0			6.0		5.0-5.1		4.1		
	10.0	26.0	31.0	6.1		6.0-6.1		4.2-4.3	7.0	
Current	11.0	27.0	32.0	7.0	19.0	7.0	5.0-7.0	4.4	10.0	10.0
Near future		28.0	33.0		20.0					
Farther future		29.0	34.0		21.0					
3 versions ahead		30.0	35.0							

Figura 8.3: Suporte ao postMessage nos browsers no início de 2014

Você pode consultar mais sobre a API na MDN:
https://developer.mozilla.org/en-US/docs/Web/API/Window.postMessage

8.3 Web Notifications

Web Notification é mais uma entre as principais features do HTML5. É uma API para enviar notificações ao usuário que são exibidas pelo browser no sistema do usuário. As notificações são mostradas mesmo que o usuário esteja em outra aba ou usando outra aplicação. São usadas, portanto, para chamar a atenção do usuário para algo importante. Note, porém, que nossa página precisa estar aberta no navegador em alguma aba, senão não temos como enviar a notificação.

A API já está em *working draft* desde 2012 e os browsers mais modernos, como Firefox e Chrome já dão suporte.

A API é bem simples e, como várias outras (Geolocation, WebRTC), é necessário pedir permissão ao usuário para poder utilizá-la. A API nos disponibiliza o objeto `Notification` e, para pedir permissão de uso ao usuário, utilizamos o método `requestPermission`:

```
Notification.requestPermission(function(perm) {
    // trata permissão
});
```

O método `requestPermission` recebe uma função de callback como parâmetro e devolve o STATUS da requisição de permissão. Este STATUS pode ter os seguintes valores: `default`, `denied` e `granted`.

Os valores `default` e `denied` significam praticamente a mesma coisa, que você não terá acesso ao recurso. A diferença está no status `denied`, que indica que o usuário explicitamente clicou na opção "negar". Podemos verificar se temos acesso:

```
Notification.requestPermission(function(perm) {
    if (permission === "granted") {
      // temos autorização para mostrar notificações
    }
});
```

Após pedir permissão, basta instanciar um objeto `Notification`:

```
var notification = new Notification("Casa do Código", {
    icon: "icon.png",
    body: "Novo livro na casa do cógigo disponível"
});
```

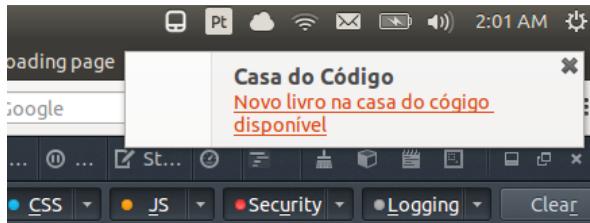



Figura 8.4: Exemplo de notificação mostrada em um Firefox no Linux

O primeiro parâmetro, e único obrigatório, é o título da notificação. O segundo parâmetro é um objeto literal com algumas opções, dentre elas `icon` — para definir um ícone para a notificação — e `body` — que define o texto da notificação.

A notificação ainda possui 4 eventos: `click`, `show`, `error` e `close`. Estes callbacks nos dão mais poder sobre cada notificação. Podemos tomar decisões específicas para cada um dos eventos.

Por exemplo, mostrar uma notificação de novo e-mail que, quando clicada, exibe o e-mail para o usuário (o Gmail faz exatamente isso):

```
var notification = new Notification("Novo Email!", {
    icon: "icon.png",
    body: "Por favor entre em contato."
});
```

```
notification.addEventListener('click',function () {
  exibeEmailNaJanela();
});
```

O interessante destes callbacks é que, mesmo que o usuário não esteja na página, ao clicar na notificação, alguma ação pode ser executada em background na aplicação.

Um detalhe importante é que, por motivos de segurança, somente uma ação explícita do usuário pode requisitar permissão para notificação. Por ação explícita, entenda alguma interação com mouse ou teclado, como clique num botão, por exemplo. Depois de aprovadas, as notificações podem ser disparadas sem ação direta.

Atualmente, apenas as versões mais novas dos browsers Firefox, Safari e Google Chrome suportam WebNotifications. Mas com a especificação se tornando um padrão oficial, devemos ver suporte nos outros navegadores em breve.

Web Notifications - Working Draft

Method of alerting the user outside of a web page by displaying notifications (that do not require interaction by the user).

Usage stats:	Global
Support:	49.37%
Partial support:	1.32%
Total:	50.69%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
								2.1		
								2.2		
						3.2		2.3		
						4.0-4.1		3.0		
	8.0			5.1		4.2-4.3		4.0		
	9.0			6.0		5.0-5.1		4.1		
	10.0	26.0	31.0	6.1		6.0-6.1		4.2-4.3	7.0	
Current	11.0	27.0	32.0	7.0	19.0	7.0	5.0-7.0	4.4 webkit	10.0	10.0
Near future		28.0	33.0		20.0					
Farther future		29.0	34.0		21.0					
3 versions ahead		30.0	35.0							

Figura 8.5: Compatibilidade das Notificações no início de 2014.

8.4 History API

Há bastante tempo os browsers têm a capacidade de manipular o histórico de navegação do usuário. Mas apenas interações básicas, como voltar e avançar eram possíveis.

A boa nova é que a API agora nos dá o poder de **adicionar entradas ao histórico**, sem causar o recarregamento da página. Além disso, a API possui um evento que é disparado a cada nova entrada ou remoção. Mark Pilgrim, em seu excelente *Dive Into HTML5*, falou sobre a HTML5 history API:

"Por que você iria manipular manualmente a barra de endereços do navegador? Afinal, um simples link pode navegar até uma nova URL; essa é a forma que a Web tem funcionado nos últimos 20 anos. E isso vai continuar funcionando desta forma. Esta API não tenta revolucionar a Web. Muito pelo contrário. Nos últimos anos, os desenvolvedores Web têm encontrado novas e excitantes formas de revolucionar a Web sem a ajuda de padrões emergentes. A API de histórico da HTML5 foi na verdade criada para garantir que as URLs continuem sendo úteis em aplicações Web com scripts pesados.

Vamos entender melhor o poder dessa API.

Conhecendo a history API

Como mencionado anteriormente, a API já existe há um bom tempo. Voltar e avançar para um determinado ponto no histórico do navegador é uma tarefa simples:

```
/* Voltar */
window.history.back();

/* Avançar */
window.history.foward();
```

Além de voltar e avançar explicitamente para o ponto anterior e posterior, é possível navegar de maneira dinâmica para um ponto do histórico:

```
/* Volta 2 passos no histórico */
window.history.go(-2);

/* Avança 3 passos no histórico */
window.history.go(3);
```

Ainda é possível saber o número de entradas no histórico, através da propriedade `length`:

```
/* Tamanho do histórico */
console.log(window.history.length);
```

Esses recursos funcionam nos browsers há muitos anos. Mas o HTML5 trouxe novidades importantes, que são o foco da nossa discussão.

Por que manipular o histórico?

Os casos de uso para as novidades na API de histórico vieram com a evolução das aplicações Web e o uso de Ajax. Nesses cenários, é comum que interações na página causem requisições Ajax ao invés de navegar para outra página nova. Essas interações modificam o estado da página mas, para deixar a eperiência do usuário mais suave, optamos por evitar o *refresh* e usar Ajax.

Um exemplo prático é uma aplicação Web como o Gmail. Na caixa de entrada, vemos a lista de e-mails. Ao clicar em um deles, é exibido o texto daquele e-mail. Toda essa interação é feita via Ajax (para obter os dados do e-mail) e não causa navegação e refresh da tela.

Mas qual o problema que a History API tentou resolver nesse caso? Quando não navegamos para uma nova página mas apenas requisitamos algo via Ajax, a **URL na barra de endereços não muda**. A URL só muda quando há navegação, e o Ajax quebra isso. O ruim de não mudar a URL é que o usuário não pode linkar, favoritar ou compartilhar uma tela intermediária da aplicação. No exemplo do Gmail, seria interessante poder guardar nos favoritos um link para um e-mail específico.

A History API trouxe, portanto, uma maneira de criar novas entradas no histórico do usuário via código JavaScript. Efetivamente, permitindo que alteremos a URL na barra de endereços e suportemos o botão de Voltar mesmo em aplicações que usam Ajax em vez de navegações comuns.

Novos métodos e o evento popstate

Os novos métodos da history API são: `history.pushState()` e `history.replaceState()`. Ambos métodos permitem adicionar e manipular entradas no histórico.

pushState

O método `pushState` adiciona uma entrada no histórico sem causar navegação de páginas. Recebe como parâmetros:

- **state object**: objeto que será associado a nova entrada;
- **title**: alguns browsers ignoram este parâmetro, mas seu objetivo é definir o título do estado;
- **URL**: a nova URL.

Vamos a um exemplo prático: Suponha que estamos no site da Casa do Código http://www.casadocodigo.com.br. Ao clicar em um livro, digamos que seja o livro *"Coletânea FrontEnd"*, vamos carregar o conteúdo via Ajax e exibi-lo na página. Gostaríamos que a URL mudasse e o histórico suportasse essa interação. Vamos usar o `pushState`:

```
// carrega dados via Ajax
ajax('dados-coletanea.json');

// muda historico e URL
var stateObj = { livro: "Coletânea FrontEnd" };
history.pushState(stateObj, "Coletânea FrontEnd",
                "products/livro-coletanea-frontend");
```

O resultado do click será a adição de uma nova entrada no histórico. A barra da URL mudará seu conteúdo para http://www.casadocodigo.com.br/products/livro-coletanea-frontend sem causar a requisção ao backend da aplicação. Em outras palavras, a página não irá recarregar.

replaceState O método `replaceState` funciona exatamente da mesma maneira que o método `pushState`, exceto pelo fato de que este altera o valor atual do histórico, em vez de criar um novo.

popstate A grande vantagem de se inserir novas entradas no histórico é permitir que o usuário navegue com Voltar e Avançar. Com isso, podemos simular a navegação entre páginas usando, por exemplo, nossos conteúdos com Ajax.

O evento `popstate` é disparado toda vez que o histórico muda conforme o usuário navega nele. Note que somente com alguma ação como o click no botão voltar, ou a chamada a alguns dos métodos de navegação pelo histórico, fará o disparo do evento. Chamadas aos métodos `pushState` e `replaceState` não disparam o evento `popstate`.

Podemos escutar o evento e recuperar o estado do histórico:

```
window.addEventListener('popstate', function(event) {
    console.log(JSON.stringify(event.state));
});
```

Supondo que este código está rodando juntamente com o exemplo anterior, o resultado seria:

```
"{"livro":"Coletânea FrontEnd"}"
```

No caso da aplicação Ajax, podemos implementar um `popstate` que refaça a requisição Ajax para a página em que o usuário navegou. Algo assim:

```
window.addEventListener('popstate', function(event) {
    var dados = JSON.stringify(event.state);
    carregaDadosLivroAjax(dados.livro);
});
```

Atualmente, a grande maioria dos browsers suportam os novos métodos e evento da history API.

\# Session history management - Candidate Recommendation

Method of manipulating the user's browser's session history in JavaScript using history.pushState, history.replaceState and the popstate event

Usage stats:	Global
Support:	73.36%
Partial support:	3.68%
Total:	77.04%

Show all versions	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	IE Mobile
								2.1		
								2.2		
						3.2		2.3		
						4.0-4.1		3.0		
	8.0					4.2-4.3		4.0		
	9.0		31.0			5.0-5.1		4.1		
	10.0	26.0	32.0			6.0-6.1		4.2-4.3	7.0	
Current	11.0	27.0	33.0	7.0	19.0	7.0	5.0-7.0	4.4	10.0	10.0
Near future		28.0	34.0		20.0					
Farther future		29.0	35.0		21.0					
3 versions ahead		30.0	36.0							

Notes | Known issues (1) | Resources (7) | Feedback | Edit on GitHub

Older iOS versions and Android 4.0.4 claim support, but implementation is too buggy to be useful. Partial support in other Safari browsers refers to other buggy behavior.

Figura 8.6: Suporte a History API nos browsers no início de 2014

Algumas referências adicionais:

http://www.whatwg.org/specs/web-apps/current-work/multipage/history.html

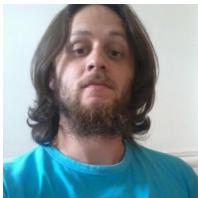
http://diveintohtml5.info/history.html

https://developer.mozilla.org/docs/Web/Guide/API/DOM/Manipulating_the_browser_history

8.5 Conclusão

As novas APIs HTML5 nos proporcionam uma gama de possibilidades que antes eram difíceis ou até mesmo impossíveis de se implementar em nossas aplicações Web.

WebStorage nos permite armazenar dados de forma padronizada e simples nos browsers sem a dependência de um sistema backend ou uso de cookies. Esta simples API pode nos trazer desde ganhos de performance em aplicações críticas até economia de recurso computacional no servidor, ou simplesmente armazenar pequenas informações relevantes sobre os usuários.

A API de postMessage nos possibilita trafegar dados *cross-domain* de forma segura e controlada, eliminando a necessidade de uma camada de proxy, ou tecnologias intermediárias para fazer o serviço.

Com *WebNotifications* podemos tornar nossas aplicações mais interativas, mesmo que o usuário não esteja presente (no momento) em nossa aplicação. Esta feature permite que a aplicação envie mensagens em forma de notificação para o usuário, até mesmo se ele estiver com foco em outra aplicação, ou em outro software, que não o browser. WebNotifications exibe notificações no nível do sistema operacional, possibilitando que o usuário obtenha informações relevantes mesmo executando outras tarefas.

Os novos métodos da HTML5 history API `pushState()` e `replaceState()`, juntamente com o evento popstate, nos proporcionam uma maneira padronizada de manipular o histórico do browser, fazendo com que nossas aplicações sejam muito mais dinâmicas para o usuário final. O uso adequado da HTML5 history API pode gerar economia de recursos no servidor, assim como melhorar a experiência do usuário.

Juntas, estas e outras novas APIs do HTML5 fazem da Web a mais poderosa plataforma de desenvolvimento de software. Seja criativo e estude os prós e contras, que o sucesso da sua aplicação estará garantido.

Sobre o autor

Jaydson Gomes é um entusiasta de tecnologias front-end, principalmente na linguagem que é um dos pilares da Web, o JavaScript. Ele possui 10 anos de experiência na área, tendo trabalhado em diversas empresas, sempre como desenvolvedor Web. Atualmente, trabalha em um dos maiores sites de notícias da América Latina, na equipe de arquititetura, criando plataformas e soluções escaláveis, com foco em front-end. Jaydson também é organizador de um dos maiores eventos de JavaScript do planeta, a BrazilJS Conf, além de organizar eventos regionais, como o RSJS e o FrontInPoa.

Capítulo 9

As APIs de acesso a dispositivos do HTML5

Desde seu nascimento, a *web* vem se tornando uma plataforma cada vez mais ubíqua e presente na vida das pessoas. Cada vez mais aplicativos, sistemas e mercados inteiros estão migrando para a web e é inegável que, apesar de ela ser uma plataforma em constante evolução, promete ser ainda melhor — o que provoca em nós, desenvolvedores, um sentimento otimista constante e nos dá a certeza de que estamos no lugar e momento certos.

Este capítulo apresenta um dos grandes grupos do HTML5, denominado de **acesso a dispositivos**. Esse grupo define várias especificações que permitem que as aplicações web possam fornecer funcionalidades e experiências mais ricas e cientes das capacidades de seus dispositivos.

Seu navegador não é mais o mesmo

Com a chegada do HTML5, o navegador web transcendeu suas origens, e tornou-se algo muito além daquele *IE6 da sua tia*. Podemos dizer que ele está cada vez mais "sensível" aos seus arredores.

"Se as portas da percepção estivessem abertas, tudo apareceria para o homem do jeito que realmente é: infinito."
– William Blake

Parafraseando William Blake, se as portas da percepção representam um filtro que impede os seres humanos de enxergar a realidade em toda sua plenitude, então podemos dizer que o HTML5 representa a abertura das portas da percepção para os navegadores. Antes do HTML5, os navegadores não "tinham ideia" da existência de vários dispositivos, mas aos poucos, à medida que novas APIs surgem, o navegador está cada vez mais "ciente" dessas informações — o que abre um enorme leque de possibilidades a respeito do que pode ou não ser feito na web.

9.1 Dispositivos e suas APIs

Um dispositivo é (quase sempre) representado por um equipamento de hardware especial, do qual o computador deve estar munido. Por exemplo, a câmera representa um sensor (hardware) dotado de mecanismos que capturam imagens em tempo real, ou seja, o equipamento físico composto por sua lente e componentes restantes.

Inovações incríveis de acesso a dispositivos ainda estão sendo desenvolvidas e implementadas, mas já podemos usufruir de algumas delas. Mais especificamente, neste capítulo, é mostrado como obter os seguintes dados através dos seus respectivos dispositivos:

- **Vídeo** via câmera;
- **Áudio** via microfone;
- **Localização geográfica** via GPS;o
- **Aceleração** em relação aos eixos ortogonais via acelerômetro;

- **Orientação** em relação aos eixos ortogonais via giroscópio;
- Informações sobre **carga e status** da bateria;
- Controle de **vibrações** do aparelho;
- **Quantidade de luz** do ambiente via sensor de iluminação.

Para cada dispositivo, há uma API que define tudo o que pode ser feito com ele.

Evolução das APIs

É importante ressaltar que essas APIs não pararam de evoluir e que, assim como qualquer coisa no mundo maravilhoso da web, futuramente elas poderão ter novos recursos — que não são abordados neste livro.

Sobre prefixos e testes de suporte nos navegadores

Durante todo este capítulo, não serão considerados os prefixos de navegadores específicos, como `moz`, `webkit`, `ms` e `o`. Por exemplo, a API *getUserMedia* será referenciada como apenas `getUserMedia` e não como `mozGetUserMedia` ou `webkitGetUserMedia`. O mesmo serve para todas as outras APIs.

Embora os prefixos tendam a desaparecer, é importante que o desenvolvedor fique atento e teste pelas suas existências antes de tentar utilizar as APIs descritas.

Durante todo o capítulo, também será considerado que o navegador suporta completamente as APIs apresentadas, mas antes de utilizá-las no mundo real, é aconselhável que seja feito um teste pela sua existência antes de qualquer coisa. Um método que pode ser utilizado para testar o suporte a uma API, inclusive considerando os prefixos é mostrado a seguir.

```
var apiPinkFloyd = window.apiPinkFloyd || window.webkitApiPinkFloyd ||
                   window.mozApiPinkFloyd || window.msApiPinkFloyd ||
                   window.oApiPinkFloyd;

if(apiPinkFloyd !== undefined){
    // tem suporte
}
```

Nesse trecho, procuramos pela existência de `apiPinkFloyd` com ou sem qualquer um dos prefixos citados. `apiPinkFloyd` pertence ao objeto global `window`, mas poderia pertencer a outro objeto qualquer, como por exemplo, ao objeto global `navigator`.

9.2 Como o navegador acessa os dispositivos

Por *acesso a dispositivo*, entende-se controlar ou acessar as informações que um dado dispositivo pode fornecer. As capacidades de controle ou acesso a informações variam de acordo com o dispositivo em questão. Por exemplo, no caso de uma câmera (geralmente uma *webcam*), podemos acessar seu vídeo capturado, no entanto **ainda** não temos o poder de controlar foco, *zoom* ou abertura do diafragma da câmera — o sistema operacional se encarrega totalmente dessas tarefas de baixo nível.

Quem define o que pode ou não ser realizado com cada dispositivo é sua API. Essa API é definida como um padrão, dessa maneira diferentes navegadores podem implementar a mesma API de maneira que sua utilização seja exatamente da igual em todos os navegadores que a implementam.

Mas como, de fato, um navegador acessa um dispositivo físico? A resposta é muito simples: **ele não acessa**. Pelo menos não de uma maneira direta. Quem faz esse acesso de baixo nível é o SO (sistema operacional), que possui os *drivers* necessários para se comunicar com o dispositivo.

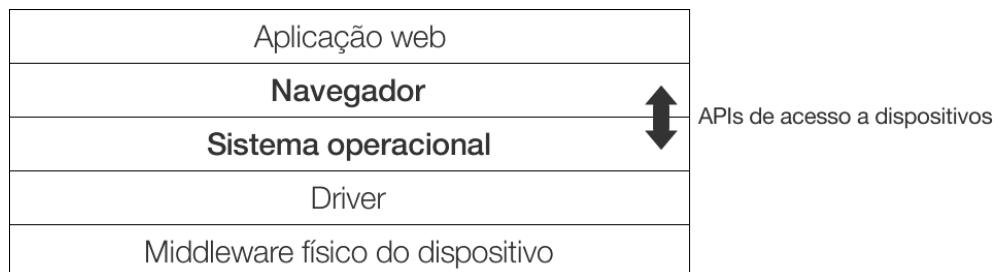



Figura 9.1: Camadas de software envolvidas no acesso a um dispositivo.

A figura anterior ilustra as camadas de software que intermedeiam a comunicação desde uma aplicação web até o software controlador do dispositivo físico (*middleware*) do hardware. Mas o que nos interessa é apenas a comunicação entre o navegador e o SO, que é intermediada pelas APIs de acesso aos dispositivos. Ou seja, note que não importa como cada SO se comunica com seus dispositivos, o desenvolvedor apenas precisa lidar com as APIs do navegador.

9.3 Câmera e microfone

Começaremos nossa turnê pelas APIs de acesso aos dispositivos mais óbvios deste capítulo: a câmera e o microfone. A possibilidade de capturar áudio e vídeo na web, por muito tempo, permaneceu apenas como mais um objeto de desejo para os desenvolvedores. Devido a uma necessidade cada vez mais crescente, isso tornou-se possível através do antigo *Macromedia Flash*, que provia essas capacidades. Com a chegada do HTML5, finalmente tornou-se possível ter acesso esses tipos de mídia, considerados tão triviais hoje em dia.

API de mídias do usuário

A API para acessar a câmera é a mesma para acessar o microfone, tornando sua utilização realmente muito simples — trata-se do método `getUserMedia()`, do objeto global `navigator`.

Por questões de privacidade, quando o método `getUserMedia()` for invocado, o navegador irá **pedir permissão** ao usuário para acessar sua câmera ou microfone — dependendo do que for requisitado.

Figura 9.2: *Prompt* do navegador Google Chrome pedindo permissão para acessar a câmera do usuário.

O método `navigator.getUserMedia()` possui 3 parâmetros:

1) **Opções**: um objeto que contém as opções de mídia que se deseja capturar. Cada opção é uma chave do objeto. Existem duas opções possíveis: `video` e `audio`, que podem ser atribuídas com um valor booleano.

2) **Callback de sucesso**: função que será executada caso o usuário libere o acesso ao dispositivo requisitado. É aqui que faremos algo com a mídia fornecida pela câmera e/ou microfone. Essa função tem apenas 1 parâmetro, que representa a *stream* de mídia fornecida.

3) **Callback de falha**: função que será executada caso o usuário **não** libere o acesso ao dispositivo requisitado ou se a mídia requisitada não for suportada. Essa função tem apenas 1 parâmetro, cujo argumento é um objeto que representa o erro, que pode ser dos tipos:

   - `PermissionDeniedError`: o usuário não permitiu o acesso ao dispositivo.
   - `NotSupportedError`: alguma mídia especificada nas opções não é suportada.
   - `MandatoryUnsatisfiedError`: nenhuma fonte dentre as mídias especificadas encontrada.

Alguns exemplos variando as opções de áudio e vídeo:

```
// requisita acesso ao microfone apenas (áudio)
navigator.getUserMedia({audio: true}, liberado, recusado);

// requisita acesso à câmera apenas (vídeo mudo)
navigator.getUserMedia({video: true}, liberado, recusado);

// requisita acesso à câmera e microfone (vídeo com áudio)
navigator.getUserMedia({video: true, audio: true}, liberado, recusado);
```

Se o usuário liberar o acesso, então a função `liberado` será disparada.

```
var liberado = function(stream){
    // uhull a stream é nossa
};
```

A partir daqui, as possibilidades são imensas, faça o que desejar a *stream* de mídia obtida.

Exibindo o vídeo da câmera

Para simplesmente capturar um vídeo e exibi-lo na tela, pode-se utilizar o elemento `<video>` do HTML5:

```
navigator.getUserMedia({video: true}, function(stream){
    // passando a stream como source do elemento video
    var video = document.querySelector('video');
```

```
    video.src = window.URL.createObjectURL(stream);
    video.play();
}, function(error){
    console.log('Ocorreu um erro: ' + error.message);
});
```

Note que, para esse trecho funcionar corretamente, deve haver um elemento `<video>` incluso no código HTML.

Brincando com vídeo da câmera no canvas

Com o elemento `<canvas>`, podemos manipular ou realizar qualquer tipo de processamento gráfico a um vídeo capturado, já que a API do *canvas* permite manipulação dos seus dados a nível de *pixel*. A ideia é jogar cada quadro (*frame*) do vídeo para o canvas por vez e, assim, realizar a manipulação gráfica desejada.

```
navigator.getUserMedia({video: true}, function(stream){
    // passando a stream como source do elemento video
    var video = document.querySelector('video');
    video.src = window.URL.createObjectURL(stream);
    video.play();

    // configurando os canvas
    var canvas = document.querySelector('canvas#canvas');
    var canvas2 = document.querySelector('canvas#canvas2');
    canvas.width = canvas2.width = 640;
    canvas.height = canvas2.width = 480;
    var ctx = canvas.getContext('2d');
    var ctx2 = canvas2.getContext('2d');

    var desenhaFrame = function(){
        window.requestAnimationFrame(desenhaFrame);
        ctx2.drawImage(video, 0, 0);
        var frame = manipulaFrame(canvas2, ctx2);
        ctx.putImageData(frame, 0, 0);
    };

    desenhaFrame();

}, function(error){
    console.log('Ocorreu um erro: ' + error.message);
});
```

No trecho anterior, estamos desenhando quadro a quadro do vídeo no canvas com a frequência estabelecida pela chamada `requestAnimationFrame`. A cada execução, manipulamos o quadro atual do vídeo com a função `manipulaFrame()`. Note que, para o trecho funcionar corretamente, deve haver um elemento `<video>` e dois elementos `<canvas>` (`#canvas` e `#canvas2`) inclusos no código HTML. Nesse exemplo estamos utilizando mais um canvas apeanas para manipular os quadros, e por isso deve permanecer invisível.

A função `manipulaFrame` pode realizar qualquer tipo de processamento na imagem, como por exemplo, converter uma imagem colorida para preto e branco. Seu pseudocódigo deve se parecer como o seguinte:

```
// função que realiza o processamento gráfico deseja na imagem
var manipulaFrame = function(canvas, ctx){
    // captura frame
    // manipula frame
    // retorna frame
};
```

Como exemplo, veja como ficaria o cálculo da imagem em preto e branco:

```
// função que realiza o processamento gráfico deseja na imagem
var manipulaFrame = function(canvas, ctx){
    // captura frame
    var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
    var pxs = imageData.data;

    // manipula frame
    for (i = 0; i < pxs.length; i+=4) {
        // converte as cores RGB do pixel em escala cinza
        var grayscale = pxs[i] * .3 + pxs[i+1] * .59 + pxs[i+2] * .11;
        pxs[i] = pxs[i+1] = pxs[i+2] = grayscale;
    }

    // retorna frame
    imageData.data = pxs;
    return imageData;
};
```

Veja esse código rodando online: http://jsfiddle.net/Q78yM/2/

Por ser um processamento que precisa ser feito em tempo real e a cada quadro obtido do vídeo capturado, pode consumir bastante os recursos de processamento de

um computador e impactar no desempenho da aplicação, então é necessário bastante cuidado com esse tipo de experimento.

Tocando o áudio do microfone

Para simplesmente capturar um áudio e tocá-lo, pode-se utilizar o elemento `<audio>` do HTML5, e exatamente da mesma maneira como no exemplo do vídeo, usamos a *stream* de áudio de forma simplificada:

```
navigator.getUserMedia({audio: true}, function(stream){
    // passando a stream como source do elemento audio
    var audio = document.querySelector('audio');
    audio.src = window.URL.createObjectURL(stream);
    audio.play();
}, function(error){
    console.log('Ocorreu um erro: ' + error.message);
});
```

Note que, para o trecho acima funcionar corretamente, deve haver um elemento `<audio>` incluso no código HTML.

Manipulando áudio do microfone

De maneira semelhante ao canvas para o vídeo, podemos utilizar a API **WebAudio** para realizar uma manipulação no áudio capturado pelo microfone. Pode-se, por exemplo, transformar a fala de alguém numa voz robótica. As possibilidades são infindáveis, pois a API WebAudio é bastante extensa e cheia de recursos. No entanto, não abordaremos sua utilização aqui.

9.4 Geolocalização

Uma das primeiras APIs de acesso a dispositivos a surgir no HTML5 foi a API de geolocalização. Com ela, passou a ser possível obter a posição do usuário em relação à superfície terrestre, ou seja, um ponto específico em nosso planeta. Geolocalização é uma informação que tem se tornado cada vez mais importante nos dias atuais.

A impotância da geolocalização

Hoje em dia, existem até redes sociais que têm como foco a localização dos seus usuários e de lugares que esses podem visitar — isso para não mencionar aplicações

de navegação em mapas e até mesmo jogos.

Além das utilidades mais comuns que acabaram de ser mencionadas, a geolocalização pode servir para muitas outras coisas importantes. Considerando a natureza contextual dessa informação, ela pode ser usada para melhorar, por exemplo, a busca e a segurança em variadas aplicações.

Pedindo uma pizza com sua localização

Sistemas de busca na web podem utilizar geolocalização para otimizar os resultados de busca para seus utilizadores. Isso já acontece há um bom tempo no Google.

Por exemplo, ao realizar uma busca por "pizzaria", o Google retornará primeiramente as pizzarias que considera ser as mais relevantes para você — e há uma grande chance de que as mais relevantes sejam as que estão mais perto de você, ou seja, de sua localização. Se o usuário estiver localizado no Rio de Janeiro ao realizar a busca, então o resultado da busca trará primeiramente as pizzarias do Rio de Janeiro, e logo depois começará a trazer outras menos relevantes, muito provavelmente no âmbito do estado (RJ) e depois por país (BR) — o que vem muito a calhar, pois, se você estiver procurando por uma pizzaria, deseja saber sobre as pizzarias próximas de você ou sobre as do **mundo inteiro**?

O Google também exibe uma miniatura de mapa (*snippet*) na lateral, mostrando um mapa local com as pizzarias marcadas em suas devidas posições. A imagem a seguir mostra a tela de um resultado de busca para o termo "pizzaria" feita no Rio de Janeiro.

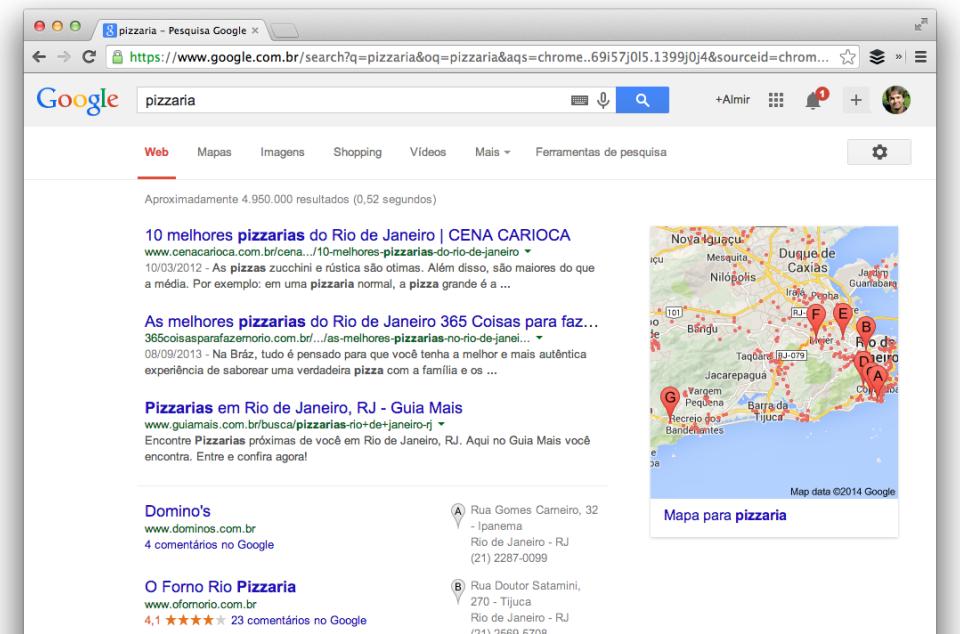



Figura 9.3: Resultado de busca no Google com as pizzarias mais badaladas do Rio de Janeiro.

Nesse caso, o Google não está utilizando a API de Geolocalização e sim fazendo uma identificação automática através do IP do usuário (detalhes sobre isso mais à frente). Mas esse caso serve para ilustrar muito bem os ganhos que podem ser obtidos com o uso dessa informação.

O Google também nos dá a possibilidade de modificar manualmente o contexto da localização utilizada em suas buscas (vide figura 9.4). Note que ao clicar na opção *Ferramentas de pesquisa*, é exibida a localização atual — no caso, *Rio de Janeiro – RJ*, que foi identificada automaticamente —, e ao clicar na localização, é exibida uma caixa com um campo textual onde se pode especificar outra localidade — e dessa maneira os resultados da busca sofrerão alterações.

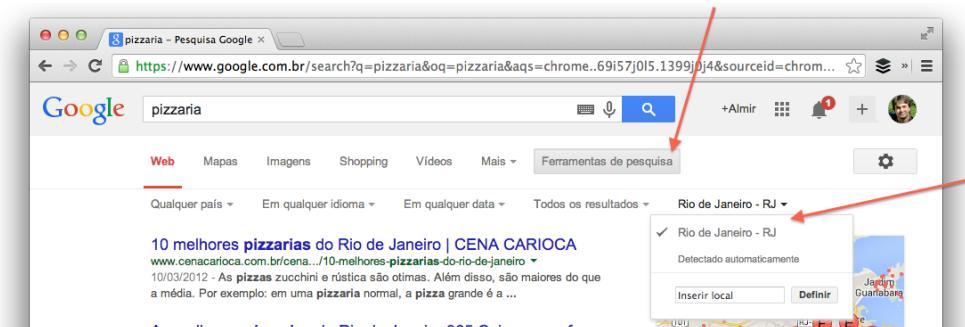



Figura 9.4: Opção de modificar a localidade para a busca no Google.

Localização pode significar mais segurança

Aplicações críticas que precisam ter um bom nível de segurança também utilizam-se da localização para melhorar nesse quesito. Bons exemplos são aplicações de *e-mail* e *bankline*, pois a segurança desses dois tipos de serviços é constantemente ameaçada. Para ambos, pode ser extremamente importante, por exemplo, manter um histórico de onde seus usuários acessaram suas contas e, com isso, tentar identificar quando a conta de um usuário é acessada (invadida) por um terceiro, que muito provavelmente esteja localizado num local muito diferente — como no exterior. Com isso, a aplicação pode requisitar uma identificação mais restrita ao usuário que tentar acessar uma conta de uma localização muito diferentes das anteriores.

Coordenadas geográficas

O globo é dividido latitude e longitudinalmente. Imagine como se fosse um plano cartesiano convencional, onde o espaço é dividido entre medidas ao logo dos eixos *x* e *y* (e opcionalmente *z*, caso seja um sistema tridimensional). Para representar um ponto nesse sistema, utilizamos uma dupla (`[x, y]`) ou uma tripla (`[x, y, z]` se for tridimensional).

A geolocalização é uma informação expressa em **coordenadas geográficas**, que similarmente a um ponto do plano cartesiano, são compostas de de 2 partes obrigatoriamente: **latitude** e **longitude**. Há também uma terceira informação (opcional)

que pode compor uma coordenada geográfica: a **altitude** (similar ao *z* do plano cartesiano).

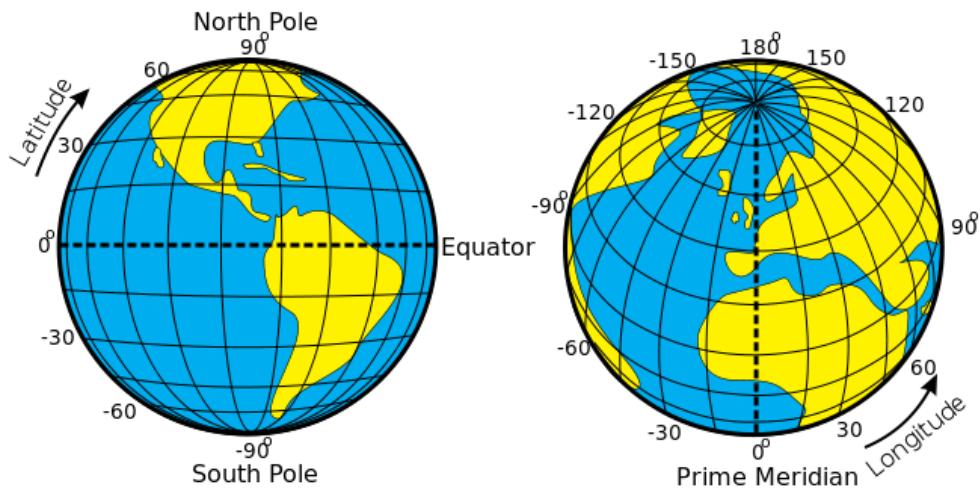



Figura 9.5: Latitude e longitude na Terra. Fonte: Wikipedia.

- **Latitude**: coordenada do eixo norte-sul da Terra. Expressa em graus.
- **Longitude**: coordenada do eixo leste-oeste da Terra. Expressa em graus.
- **Altitude**: altitude do ponto (composto pela latitude e longitude) em relação ao nível do mar. Expressa em metros. Opcional.

Como o navegador obtém as coordenadas do usuário

Há diversas maneiras de se obter as coordenadas de localização do usuário. O navegador — através do sistema operacional — tenta obter essa informação de 4 maneiras principais.

- **GPS**: se o computador (ou dispositivo móvel) for equipado com GPS, tenta obter a localização através do sistema de GPS (satélites). Esse é o método de melhor resultado, ou seja, a localização obtida tem uma melhor precisão em relação aos outros métodos. É comumente presente em dispositivos móveis.

- **Informação Wi-Fi**: se estiver conectado a uma rede Wi-Fi, tenta obter a localização a partir de serviços online que informam a localização (previamente cadastrada) do roteador em que se está conectado. Exemplo de serviço: Skyhook (skyhookwireless.com) — utilizado pelo iOS da Apple. Esse método consegue ser bem preciso, mas não tanto quanto o GPS.

- **Antenas de celular**: se estiver conectado a um serviço de celular, realiza-se a triangulação das antenas de celular mais próximas para tentar definir uma área comum. Esse método é menos preciso que o anterior, geralmente conseguindo apenas delimitar uma área com uma margem de erro bem maior que o GPS e Wi-Fi.

- **IP *lookup***: se estiver conectado à internet, tenta obter a localização a partir de serviços de pesquisa de endereços IP. Exemplo: ip-adress.com. Esse método é mais comumente utilizado pelos computadores desktop (de localização fixa) conectados por meio de conexão a cabo.

Há também uma questão importante a se considerar: o tipo de dispositivo (no sentido de computador) sendo utilizado. Basicamente existem duas classificações: os de **posição fixa** e os de **posição variável**.

- **Dispositivos de posição fixa**: geralmente são computadores desktop ou laptop (depende do laptop). Não possuem GPS nem serviço de celular — a não ser que seja utilizado um modem 3G, por exemplo. O cenário mais comum é utilizar-se de informações de Wi-Fi ou apenas o IP *lookup*.

- **Dispositivos de posição variável**: geralmente são os dispositivos móveis, como *smartphones* e *tablets*. É muito comum que sejam equipados com GPS e ainda utilizem um serviço de celular. Wi-Fi também é presente, fazendo com que esse tipo de dispositivo tenha à sua disposição todos os 4 métodos de obtenção de localização.

O navegador apenas requisita para o SO a localização e o SO, por sua vez, é quem decide qual o método que fornecerá a melhor solução para o dado momento, de acordo com a disponibilidade de cada um dos 4 métodos. Alguns sistemas utilizam uma solução mista, que mesclam informações de mais de um método para produzir uma localização mais precisa.

API de geolocalização

A API de geolocalização encontra-se no objeto `geolocation` dentro de `navigator`. Então, para testar o suporte em um navegador, pode-se fazer simplesmente:

```
if('geolocation' in navigator){
    // tem suporte a geolocalização
}
```

A API é bem simples e possui apenas 3 métodos:

- `getCurrentPosition()`: obtém as informações referentes à geolocalização do usuário no dado momento;
- `watchPosition()`: observa e atualiza as informações de geolocalização do usuário, conforme o mesmo muda de localização;
- `clearWatch()`: cancela algum `watchPosition()` que esteja em andamento.

Por questões de privacidade, ao chamar os métodos `getCurrentPosition()` ou `watchPosition()`, o navegador primeiramente requisita a permissão do usuário para acessar sua geolocalização. No navegador Google Chrome, aparece a já conhecida barra de permissões, como mostrado na figura a seguir:

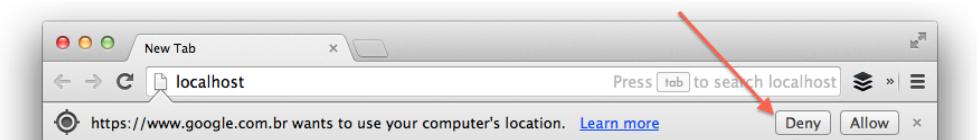



Figura 9.6: *Prompt* do navegador Google Chrome pedindo permissão para acessar a geolocalização do usuário.

Obtendo a geolocalização

Para obter a geolocalização pontual do usuário, utiliza-se o método `getCurrentPosition()`:

```
// obtem geolocalização
navigator.geolocation.getCurrentPosition(function(position){
    posicionarMapa(position.coords.latitude, position.coords.longitude);
});

// maneira mais elaborada
navigator.geolocation.getCurrentPosition(function(position){
    posicionarMapa(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log('Ocorreu um erro: ' + error.message);
}, {
    enableHighAccuracy: true,
    timeout: 10000,
    maximumAge: 0
});
```

No trecho de código anterior, as duas chamadas são bem similares, porém a segunda são definidos mais 2 argumentos de uso opcional. Os 3 parâmetros do método `getCurrentPosition()` são:

- ***Callback* de sucesso**: função de *callback* executada ao receber a reposta com as informações de geolocalização. Possui apenas 1 parâmetro, que é o objeto contendo as informações.

- ***Callback* de falha** (opcional): função de *callback* executada se houver algum erro. Possui apenas 1 parâmetro, que é um objeto contendo o tipo (`code`) e a mensagem (`message`) do erro. Os tipos de erro podem ser:

  - `PERMISSION_DENIED` (`code = 1`): usuário negou permissão de uso.
  - `POSITION_UNAVAILABLE` (`code = 2`): serviços não foram capazes de fornecer a localização no dado momento.
  - `TIMEOUT` (`code = 3`): estourou o tempo máximo tolerado.

- **Opções** (opcional): objeto com opções adicionais, que podem ser:

  - `enableHighAccuracy`: booleano que permite indicar que a localização requisitada deve ter a maior precisão possível. Isso pode deixar o processo de obtenção da localização mais lento.

- `timeout`: tempo máximo de tolerância para obtenção de localização. Expresso em milisegundos.
  - `maximumAge`: indica a possibilidade de obter-se posições previamente obtidas e *cacheadas* com uma idade máxima em milisegundos. Se for 0, então sempre requisitará novas posições (sem *cache*).

Informações obtidas

O objeto `position` que é recebido como argumento no *callback* de sucesso possui as seguintes propriedades e valores:

```
{
    timestamp: 1395810885193,
    coords: {
        latitude: -23.0068436,   // graus (-90 até 90)
        longitude: -43.3223877, // graus (-180 até 180)
        altitude: 88.43,         // metros [opcional]
        accuracy: 10,            // metros [opcional]
        altitudeAccuracy: 15,    // metros [opcional]
        heading: 271.32,         // graus (0 até 360) [opcional]
        speed: 7.64              // metros por segundo [opcional]
    }
}
```

Note que há algumas informações além das coordenadas geográficas mencionadas anteriormente. Todas essas informações a mais são opcionais e serão fornecidas dependendo se o dispositivo utilizado tem a capacidade para isso. No desktop, geralmente serão obtidas apenas as informações de latitude e longitude. Já num iPhone são obtidas todas elas, com exceção à velocidade (`coords.speed`).

- `timestamp`: momento em que a localização foi obtida.
- `coords.accuracy`: precisão (margem de erro) das coordenadas obtidas.
- `coords.altitudeAccuracy`: precisão (margem de erro) da altitude obtida.
- `coords.heading`: direção para onde o dispositivo está apontando. O valor é representado entre o intervado de 0 até 360 graus (0 representa o norte, 90 o leste, 180 o sul e 270 o oeste). Pode servir por exemplo, para implementar uma bússola.

- `coords.speed`: velocidade de deslocamento do dispositivo expressa em metros por segundo.

Observando a geolocalização

É muito comum que uma aplicação que mostre a localização do usuário utilize um mapa para isso e, geralmente, também é desejável que a posição do usuário não seja estática. Ou seja, se o usuário se deslocar com o dispositivo para outro lugar, sua posição no mapa deve ser atualizada concomitantemente.

Para alcançar tal objetivo, é preciso manter a aplicação consultando e verificando a posição constantemente. Isso poderia facilmente ser obtido fazendo-se *polling* com o método `getCurrentPosition()`, porém, sabemos que o uso de *polling* não é nada elegante do ponto de vista de engenharia de software, mandando o paradigma de desenvolvimento orientado a eventos diretamente pro espaço, além de ser constantemente associado a gambiarras (por que será?). Felizmente, não é necessário utilizarmos *polling*, pois a API provê um método específico para constantemente observar a posição do usuário: `watchPosition()`.

```
// observa geolocalização
navigator.geolocation.watchPosition(function(position){
    posicionarMapa(position.coords.latitude, position.coords.longitude);
});

// maneira mais elaborada
navigator.geolocation.watchPosition(function(position){
    posicionarMapa(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log('Ocorreu um erro: ' + error.message);
}, {
    enableHighAccuracy: true,
    timeout: 10000,
    maximumAge: 0
});
```

A parte boa desse método é que seus parâmetros são exatamente os **mesmos** que os de `getCurrentPosition()`. Note que a única diferença entre esse trecho de código e o anterior é o nome dos métodos. Os *callbacks* de sucesso e de erro também recebem os mesmos objetos com as mesmas informações — `position` e `error`, respectivamente.

Após `watchPosition()` ser chamado, ele se manterá num estado de espera — não bloqueante, pois as respostas são assíncronas — e disparando os *callbacks* de erro ou sucesso sempre que as respostas chegarem.

Outro detalhe é que o método `watchPosition()` retorna um número identificador único (ID), que serve para que se possa realizar seu cancelamento posteriormente. Para cancelar uma "observação", por assim dizer, utiliza-se o método `clearWatch()`, que recebe apenas 1 argumento: o ID de uma "observação" previamente criada.

```
// obtendo id da observação
var watchId = navigator.geolocation.watchPosition(function(position){
    posicionarMapa(position.coords.latitude, position.coords.longitude);
});

// cancelando a observação quando clicar no botão
botaoFechar.addEventListener('click', function(){
    navigator.geolocation.clearWatch(watchId);
});
```

Mostrando a localização em um mapa

Com o intuito de exemplificar a simplicidade de uso da API de geolocalização, será demonstrado como mostrar um mapa com sua localização atual. Para desenhar o mapa desse exemplo, foi escolhida a API do Google Maps, cuja utilização é bem simples.

Começando com o HTML, temos uma marcação bem simples a seguir. O elemento `<div id="mapa">` é utilizado pela API do Google Maps monte o mapa dentro do mesmo. Depois são inclusos os *scripts* necessários: a API do Google Maps e a nossa aplicação.

```
<!doctype html>
<html>
<head>
    <title>Mapa</title>
</head>
<body>
    <div id="mapa"></div>
    <script type="text/javascript"
        src="https://maps.google.com/maps/api/js?sensor=true"></script>
    <script type="text/javascript" src="meu_mapa.js"></script>
```

```
</body>
</html>
```

O conteúdo de `meu_mapa.js` (nossa aplicação) deve se parecer com o seguinte:

```
var montarMapa = function(position){
    // objeto de localizacao do google maps
    var localizacao = new google.maps.LatLng(
        position.coords.latitude,
        position.coords.longitude
    );

    // elemento DOM onde será montado o mapa
    var elementoMapa = document.getElementById('mapa');

    // objeto de mapa do google maps
    var mapa = new google.maps.Map(elementoMapa, {
        zoom: 10,
        center: localizacao,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    });

    // adiciona um marcador para a localização no mapa
    new google.maps.Marker({
        map: mapa,
        position: localizacao,
        title: 'Você está aqui'
    });
};

navigator.geolocation.getCurrentPosition(montarMapa, function(error){
    console.log('Ocorreu um erro: ' + error.message);
});
```

Quando `getCurrentPosition()` obtém a localização, a função `montarMapa()` é chamada — que por sua vez, recebe o objeto `position` com as coordenadas usadas para construir o objeto de posição geográfica do Google Maps (`google.maps.LatLng`). O restante do *script* se encarrega apenas de montar um mapa centralizado na localização recebida e de colocar um marcador no mapa que indique visualmente essa localização.

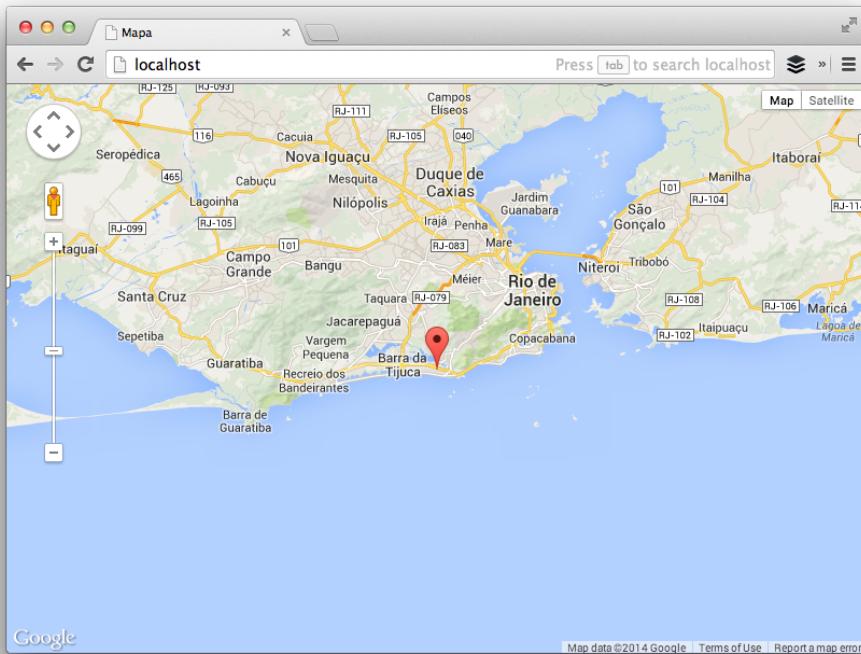



Figura 9.7: Mapa da API do Google Maps mostrando a localização atual do dispositivo.

Veja esse exemplo executando online: http://jsfiddle.net/LmbXN/

Observação: o elemento utilizado para o mapa — nesse exemplo, `<div id="mapa">` — deve possuir suas dimensões definidas (altura e largura), caso contrário o mapa não aparecerá. As dimensões podem ser definidas através de CSS (propriedades `width` e `height`).

9.5 Acelerômetro e giroscópio

O HTML5 define uma nova API para ter acesso às informações fornecidas por dispositivos de aceleração e orientação (nos dispositivos que possuem acelerômetro e giroscópio).

Praticamente todos os dispositivos móveis (smartphones e tablets) do mercado hoje em dia vêm equipados com estes sensores e o que não falta também são aplicati-

vos e jogos que se utilizam dessas habilidades. Já no desktop a coisa muda um pouco e seu uso não é muito comum, com exceção aos MacBooks, que já vêm equipados com esse tipo de hardware.

Para que serve um acelerômetro no MacBook?

O acelerômetro desempenha um papel importante nos MacBooks. Eles são usados para detectar algum movimento brusco e, a partir disso, o sistema pode se tomar uma atitude para se proteger de algo. Por exemplo, se o MacBook for solto de uma certa altura, o sistema detecta uma aceleração elevada e tenta desligar o disco rígido (HDD) do computado a tempo, antes que este seja altamente danificado por causa da queda. Não testem isso em casa.

Para testar o suporte a API:

```
if('DeviceOrientationEvent' in window){
    // seu navegador suporta DeviceOrientationEvent
}

if('DeviceMotionEvent' in window){
    // seu navegador suporta DeviceMotionEvent
}

if('CompassNeedsCalibrationEvent' in window){
    // seu navegador suporta CompassNeedsCalibrationEvent
}
```

Antes de prosseguirmos com a API e seus eventos, algo deve ficar bem claro: os **eixos do sistema de coordenadas** usados. Os eventos retornam dados referentes a cada um dos eixos X, Y e Z:

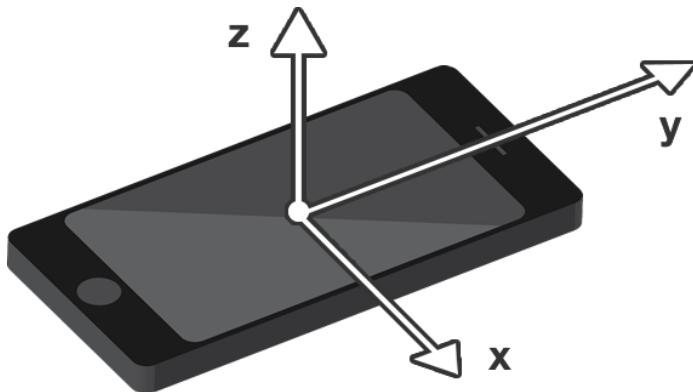



Figura 9.8: Eixos do sistema de coordenadas da API.

Onde:

- **X** é o eixo que "percorre" a tela do dispositivo de lado a lado (perpendicular a Y e Z);
- **Y** é o eixo que "percorre" a tela do dispositivo de baixo para cima (perpendicular a X e Z);
- **Z** é o eixo que completa o sistema, é representado como estivesse "saindo" da tela (perpendicular a X e Y).

Detalhe: estando em um laptop, estes eixos serão relativos ao plano do teclado, e não ao plano da tela.

A API consiste basicamente da implementação de três eventos:

- `deviceorientation`: acontece quando há variação na orientação do dispositivo;
- `devicemotion`: ocorre quando há variação na aceleração de deslocamento do dispositivo;
- `compassneedscalibration`: disparado quando é detectada a necessidade de calibar a bússola do giroscópio.

Obtendo a orientação do dispositivo

Para ter acesso às informações sobre a orientação do dispositivo, é utilizado evento `deviceorientation`. Esse evento é disparado quando há novos dados disponíveis fornecidos pelo sensor de orientação (giroscópio), ou seja, quando a orientação atual muda. Para termos acesso a esses dados, precisamos apenas definir uma função manipuladora para o evento:

```
window.addEventListener('deviceorientation', function(orientData){
    // move bola do jogo de acordo com a orientação do dispositivo
    moverBola(orientData.alpha, orientData.beta, orientData.gamma);
});
```

Todos os dados disponíveis na ocorrência do evento estarão em `orientData`. As propriedades contidas em `orientData` — além dos dados referentes à interface `Event` — são: `alpha`, `beta`, `gamma` e `absolute`.

As propriedades `alpha`, `beta` e `gamma` representam a rotação em graus em torno de cada um dos eixos Z, X e Y respectivamente, como ilustrado na figura seguinte. A propriedade `absolute` é um booleano que indica se os dados de rotação são absolutos (referentes ao solo) ou não (referentes a ele mesmo ou outra posição arbitrária inicial).

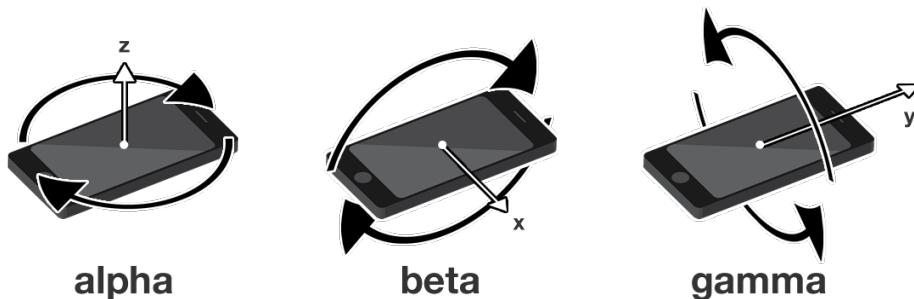



Figura 9.9: Rotações *alpha*, *beta* e *gamma* nos 3 eixos X, Y e Z.

Por exemplo, se deixarmos o smartphone do desenho em pé, ou seja, com o eixo Y apontando para cima e o eixo Z apontando para nós, *beta* será igual a 90, pois apenas realizamos uma rotação de 90° em torno do eixo X. Já se deixarmos o mesmo smartphone de cabeça para baixo, *beta* será igual a -90, pois realizamos a mesma rotação no sentido contrário.

Obtendo a aceleração do dispositivo

O evento `devicemotion` é disparado periodicamente e fornece dados sobre movimentos físicos realizados durante um certo intervalo de tempo (o intervalo entre as chamadas do próprio evento). Fornece dados sobre taxa de rotação, assim como a aceleração referente a cada um dos três eixos. Para termos acesso a esses dados, apenas definimos uma função para manipulação do evento `devicemotion`:

```
window.addEventListener('devicemotion', function(eventData){
    // faça algo de legal aqui
});
```

Os dados fornecidos por `eventData` — além dos dados referentes à interface `Event` — são: `acceleration`, `accelerationIncludingGravity` e `rotationRate`.

- `acceleration`: fornece a aceleração do dispositivo em relação a cada um dos três eixos.
  - `x`: aceleração em relação ao eixo X (expressa em m/s^2);
  - `y`: aceleração em relação ao eixo Y (expressa em m/s^2);
  - `z`: aceleração em relação ao eixo Z (expressa em m/s^2).
- `accelerationIncludingGravity`: igualmente a `acceleration`, fornece a aceleração do dispositivo em cada um dos três eixos (X, Y e Z). A diferença está no fato de o hardware ter ou não a capacidade de excluir o efeito da gravidade. Por exemplo, no caso de o dispositivo estar "deitado" e em repouso, a aceleração no eixo Z sempre terá um valor aproximado a 9.8 (gravidade terrestre).
  - `x`: aceleração (com gravidade) em relação ao eixo X (expressa em m/s^2);
  - `y`: aceleração (com gravidade) em relação ao eixo Y (expressa em m/s^2);
  - `z`: aceleração (com gravidade) em relação ao eixo Z (expressa em m/s^2).
- `rotationRate`: provê as taxas de rotação referentes a cada eixo, propriedades `alpha`, `beta` e `gamma` (exatamente os mesmos dados fornecidos pelo evento `deviceorientation`).

O evento compassneedscalibration

Como o próprio nome diz, este evento deverá ser acionado quando o navegador determinar que um acelerômetro precisa ser (re)calibrado. Isso deve ocorrer apenas se for preciso aumentar a precisão ou corrigir alguma irregularidade nos dados. Este evento possui uma ação padrão que fica a cargo de cada navegador. O navegador pode, por exemplo, mostrar detalhes ao usuário de como realizar a calibração. Também deve ser possível modificar este comportamento padrão de modo que as aplicações possam apresentar sua própria interface para essa tarefa.

9.6 Bateria

A **Battery API** permite que tenhamos acesso a informações sobre a bateria do nosso dispositivo — seja ele um dispositivo móvel (smartphone ou tablet) ou um laptop No caso de computadores desktop, não teremos esta informação disponível, a não ser que exista algum equipado com bateria.

A principal utilidade dessa API se dá na plataforma móvel, onde geralmente temos recursos bem mais escassos em relação aos outros tipos de dispositivos com os quais estamos acostumados (desktops e laptops). Um destes recursos é justamente a quantidade de energia disponível.

Sem saber qual a situação da bateria, um desenvolvedor terá que projetar uma aplicação assumindo o risco de que sempre haverá quantidade de energia suficiente para realizar seja qual for a tarefa pretendida. Isto significa que uma aplicação mal implementada — no sentido de ser ineficiente — poderá exaurir a bateria do usuário facilmente, piorando assim a experiência do usuário.

Por exemplo, imagine que desenvolvemos uma *web app* para o Firefox OS. Nossa app deverá realizar *polling*, isso é, deverá, de tempos em tempos, acessar uma determinada fonte de dados para checar se há informação nova disponível — digamos, por exemplo, que nossa app pergunta ao Facebook de 10 em 10 segundos se há algo de novo no *feed*. Em um intervalo de 1 hora, nossa aplicação fará **360 requisições** só para saber se há algo de novo. 360 requisições por hora pode facilmente comprometer a bateria de um dispositivo móvel. O ideal seria, por exemplo, fazer essas requisições de 30 em 30 segundos (baixando a quantidade de requisições por hora para 120) quando o dispositivo não estiver carregando e manter a frequência de 5 segundos para quando estiver conectado a uma fonte de energia.

Com a Battery API será possível aos desenvolvedores ter um *feedback* mais apurado em relação a este tipo de problema, o que auxiliará na tomada de decisão em

situações importantes e consequentemente melhorará a qualidade do software final.

A API da bateria

Todas as funcionalidades relacionadas a bateria estão definidas na interface `BatteryManager`, que é acessível através do objeto `navigator.battery`. Para testar o suporte:

```
if('battery' in navigator){
    // seu navegador suporta a Battery API
}
```

A API possui apenas 4 propriedades bem simples: `charging`, `chargingTime`, `dischargingTime` e `level`. Justamente pela simplicidade, cada uma delas já se autoexplica apenas pelo seu nome.

```
if(navigator.battery.charging){
    // a bateria está sendo carregada
    console.log(navigator.battery.chargingTime); // 3226s
    // faltam 53.7min para terminar de carregar
} else {
    // a bateria NÃO está sendo carregada
    console.log(navigator.battery.dischargingTime); // 5564s
    // faltam 1h32min para descarregar totalmente
}

// nível de carga
console.log(navigator.battery.level); // 0.34 = 34% de carga
```

- `level`: indica o nível de carga atual da bateria — número ponto flutuante (*float*) de 0 a 1.
- `charging`: booleano que indica se a bateria do dispositivo está ou não sendo carregada. Note que isto não é o mesmo que dizer se a bateria está (ou não) conectada a uma fonte de energia, pois temos o caso de estar ligada na fonte e 100% carregada, o que implica em dizer que o dispositivo talvez passe a utilizar a energia proveniente da fonte — isto acontece no MacBook e nos dispositivos iOS.
- `chargingTime`: fornece o tempo estimado para que a bateria carregue totalmente. Se a bateria estiver carregando — ou seja, se `charging` for `true` —,

seu valor será um número inteiro que representa o tempo em segundos. Caso contrário, será `'Infinity'` — o que faz sentido, pois se a bateria não está carregando, podemos dizer que o tempo restante estimado é infinito.

- `dischargingTime`: ao contrário de `chargingTime`, fornece o tempo estimado para que a bateria descarregue totalmente. Se a bateria não estiver carregando — quando `charging` for `false` —, seu valor será um número inteiro que representa o tempo em segundos. Caso contrário, será `'Infinity'`.

Eventos da API

Para cada uma das 4 propriedades descritas antes, há um evento associado.

- `onchargingchange`: será disparado toda vez que houver uma mudança nos *status* carregando/descarregando, ou seja, quando o valor de `charging` for modificado.
- `onchargingtimechange`: se a bateria estiver carregando, esse evento será disparado toda vez que o valor de `chargingTime` mudar.
- `ondischargingtimechange`: de maneira análoga a `onchargingtimechange`, se a bateria **não** estiver carregando, esse evento será disparado toda vez que o valor de `dischargingTime` mudar.
- `onlevelchange`: será disparado toda vez que o nível de carga da bateria variar, ou seja, quando o valor de `level` for modificado.

```
navigator.battery.addEventListener('chargingchange', function(){
    if(navigator.battery.charging){
        console.log('Bateria carregando');
    } else {
        console.log('Bateria DEScarregando');
    }
}, false);

navigator.battery.addEventListener('chargingtimechange', function(){
    var tempoRestante = navigator.battery.chargingTime / 60;
    console.log('Faltam ' + tempoRestante + ' minutos para carregar');
}, false);

navigator.battery.addEventListener('dischargingtimechange', function(){
```

```
    var tempoRestante = navigator.battery.dischargingTime / 60;
    console.log('Faltam ' + tempoRestante + ' minutos para descarregar');
}, false);

navigator.battery.addEventListener('levelchange', function(){
    var nivel = navigator.battery.level * 100;
    console.log('O nível da bateria mudou para ' + nivel + '%');
}, false);
```

9.7 Vibração

Embora ainda não seja possível provocar terremotos de escala geográfica, já podemos fazer dispositivos vibrar com HTML5. Entenda por dispositivo qualquer meio de acesso que possua um hardware específico que possibilite isso — como os dispositivos móveis, pois não faria o mínimo sentido seu computador desktop ter a habilidade de vibrar.

Muitas dessas novas APIs de acesso a dispositivos são focadas em utilidades *mobile* (assim como a Battery API da seção anterior), a Mozilla é a entidade que mais investe nessas especificações, pelo simples fato de que ela mantém o Firefox OS, cuja proposta é funcionar totalmente em cima dos padrões abertos da web.

Considerando que o mecanismo de vibração causa um simples *feedback* tátil, existem algumas utilidades que podem ser de interesse para uma aplicação móvel. As mais comuns são, por exemplo, alertar o usuário ao disparar uma notificação, mensagem ou ligação e vibrar em momentos específicos ao decorrer de um jogo (uma bomba explodindo, talvez) — causando uma maior sensação de imersão para o jogador.

Outras utilidades — não tão óbvias — poderiam ser, por exemplo, guiar um usuário portador de deficiência visual em um ambiente, onde cada tipo de vibração poderia corresponder a direcionamentos como 'esquerda', 'direita', 'trás', 'frente' etc. A partir daí vai de acordo com a criatividade de cada um.

Curiosidade

Originalmente, o nome dessa API seria *Vibrator* (vibrador), mas, por questões óbvias, preferiram mudar para *Vibration*.

API para vibração

Essa é a API mais simples deste capítulo, pois é composta de apenas 1 método super simples — `vibrate`, que reside no objeto `navigator`. Para testar o suporte:

```
if('vibrate' in navigator){
    // tann namm!
}
```

De acordo com a especificação, esse método recebe apenas um parâmetro que representa uma duração de tempo que o dispositivo deve permanecer vibrando.

```
navigator.vibrate(1000); // vibra por 1000ms (1s)
```

Esse parâmetro também pode ser uma lista representando um padrão de toques, onde cada item da lista alterna entre tempo de duração de uma vibração e o tempo de duração de uma intervalo. Com um exemplo fica mais fácil de entender:

```
navigator.vibrate([500, 1000, 800, 500, 1000]);
```

No trecho anterior, o dispositivo começará vibrando por 500ms, depois pausa por 1000ms, vibra novamente por 800ms, pausa por 500ms e termina vibrando por mais 1000ms.

Caso uma chamada de vibração já tenha sido disparada, também é possível cancelá-la passando um zero (0) ou uma lista (*array*) vazia como parâmetro.

```
navigator.vibrate(0);  // cancela qualquer vibração em execução
navigator.vibrate([]); // mesma coisa
```

9.8 Iluminação ambiente

Esse com certeza é um dos recursos mais interessantes que têm surgido dentro da área de acesso a dispositivos. Ele permite que tenhamos acesso à quantidade de luminosidade do ambiente, ou seja, podemos saber se o usuário está situado num lugar escuro, por exemplo. Logicamente, o dispositivo do usuário em questão deve ser equipado com um sensor de luz — qualquer MacBook e a maioria dos dispositivos móveis já o possuem.

Uma utilidade que pode ser interessante pode ser tornar automática uma funcionalidade bastante comum entre as aplicações de leitores de *feeds* (como o Pocket,

Digg e Readability), que é modificar o contraste do texto. Assim o usuário pode alternar entre níveis de contraste de maneira a tornar o texto mais confortável para leitura:

Figura 9.10: Contrastes disponíveis no leitor de *feeds* Pocket.

Se você estivesse à noite na cama e com as luzes apagadas, qual contraste seria o mais confortável para ler? A maioria das pessoas com certeza escolheria o da direita (com fundo escuro). Graças a essa ***Ambient Light Events* API** torna-se possível tomar decisões com base na quantidade de luz atual do ambiente.

API de iluminação

Esta nova especificação da W3C define dois novos eventos capazes de detectar mudanças na iluminação do ambiente: `onlightlevel` (*Device Light*) e `ondevicelight` (*Device Level*). Para testar o suporte:

```
if('ondevicelight' in window){
    // seu navegador dá suporte ao evento *Device Light*
}

if('onlightlevel' in window){
    // seu navegador dá suporte ao evento *Light Level*
}
```

> **A UNIDADE *LUX***
>
> *Lux* é uma unidade que serve para representar valores de quantidade de luz ambiente.

- `onlightlevel`: disparado toda vez quando houver uma mudança de estado em relação a iluminação. Existem 3 estados diferentes:
  - **dim**: o dispositivo se encontra num ambiente escuro ou com pouca iluminação. Exemplo: noite ou quarto escuro.
  - **normal**: o ambiente tem um nível ideal de luminosidade. Exemplo: sala iluminada adequadamente.
  - **bright**: o ambiente tem um nível muito claro ou até excessivo em quantidade de *lux* (mais sobre isso adiante), fazendo com que a tela adquira um aspecto meio "apagado". Exemplo: luz do dia.
- `ondevicelight`: este evento é bem similar ao `onlightlevel`, a diferença está na sua granularidade. Em vez de apenas três possíveis estados (*dim*, *normal* e *bright*), temos acesso direto ao valor de iluminação do ambiente expresso em *lux*.

```
window.addEventListener('lightlevel', function(event) {
  console.log(event.value); // dim, normal ou bright
});

window.addEventListener('devicelight', function(event) {
  console.log(event.value); // 5, 10, 78, ...
});
```

Cabe aos navegadores implementarem o intervalo em *lux* que define cada um dos três estados *dim*, *normal* e *bright*. Entretanto, a especificação da W3C recomenda que o *dim* corresponda a ambientes com iluminação menores que 50 *lux* — escuro o suficiente para que uma luz produzida por um fundo branco seja uma distração —, *normal* corresponda a um valor entre 50 e 10.000 *lux* — uma sala de escritório, o nascer ou pôr do sol em uma dia claro — e o *bright* a uma iluminação acima de 10.000 lux — algo como luz solar direta.

Um exemplo prático pode ser o que muda as cores da página de acordo com o nível de luz. O código é bem simples: podemos pegar, por exemplo, o nome do nível de luz e aplicá-lo como uma classe no HTML.

```
window.addEventListener('lightlevel', function(event) {
  document.body.className += event.value;
});
```

E, via CSS, estilizar a página de maneira diferente:

```
.dim {
    background: black;
    color: white;
}

.normal {
    background: #eee;
    color: #222;
}

.bright {
    background: white;
    color: black;
}
```

9.9 Conclusão

Os aparelhos estão equipados com cada vez mais sensores e recursos de hardware. O aparelho Samsung Galaxy S4, por exemplo, já vinha com 9 sensores, incluindo barômetro, sensor de temperatura e umidade. A web, em sua constante evolução, suporta cada vez mais esses tipos de recursos.

Aqui, discutimos alguns recursos de hardware comuns e casos práticos para usá-los. Há muitas outras possibilidades e a verdade é que os desenvolvedores estão apenas começando a explorar todo esse potencial. Inspire-se!

Sobre o autor

Almir Filho é especialista em desenvolvimento Web na Globo.com e cofundador do blog Loop Infinito (loopinfinito.com.br), onde compartilha seu conhecimento e experiências sobre front-end e pensamentos sobre o papel maluco que adquirimos no mercado como desenvolvedores. Possui graduação e mestrado em Ciência da Computação pela Universidade Federal do Maranhão, é entusiasta e extremamente interessado pelos padrões Web e produtividade. Amante do mundo e artista de sanduíches. (Twitter: @almirfilho)

Capítulo 10

Debugando sua Web App — ou, Como se estressar menos

"Assim que começamos a programar, descobrimos — para nossa surpresa — que não era tão fácil de obter programas sem erros, como tínhamos pensado. A depuração teve de ser descoberta. Lembro-me do momento exato em que eu percebi que uma grande parte da minha vida a partir de então iria ser gasta em encontrar erros em meus próprios programas."
– Maurice Wilkes, 1949

Por debugar — ou depurar — código entende-se o ato de remover *bugs*. Bugs, por sua vez, são comportamentos inesperados em um dado *software*. Nenhum programador irá escapar de um dia — ou todos os dias — debugar um código para identificar e corrigir um problema — ou bug — em um software. E terá muita sorte se o código que for debugar foi escrito por ele mesmo.

Debugar costuma ser uma tarefa mais custosa que escrever código. E debugar código JavaScript *client-side* é especialmente dificil pela falta de controle que temos

sobre o ambiente onde o código será executado. São vários sistemas operacionais, várias versões de sistemas operacionais, vários *browsers*, várias versões de browsers, várias extensões, várias resoluções de tela e diferentes velocidades de conexão. Em uma dessas permutações se esconde um bug que você terá que caçar.

Logo, dominar as ferramentas de *debug* se torna imperativo para que seja gasto menos tempo procurando e corrigindo bugs de seu software.

A origem do termo Debug

Apesar de controvérsias a respeito da origem do termo "debug", ela é comumente atribuida a Grace Hopper, uma talentosa desenvolvedora da Marinha dos E.U.A.

Enquanto trabalhava no computador Mark II da Universidade de Harvard, por volta de 1945, uma equipe de técnicos observou um comportamento estranho no painel F da máquina. Quando foram investigar o porquê do comportamento estranho, viram que havia uma mariposa morta — um inseto, ou *bug* em inglês — em um dos relés, que estava causando tal funcionamento anômalo. A equipe prontamente retirou a mariposa e Grace Hopper comentou que eles estavam "debugando" o sistema.

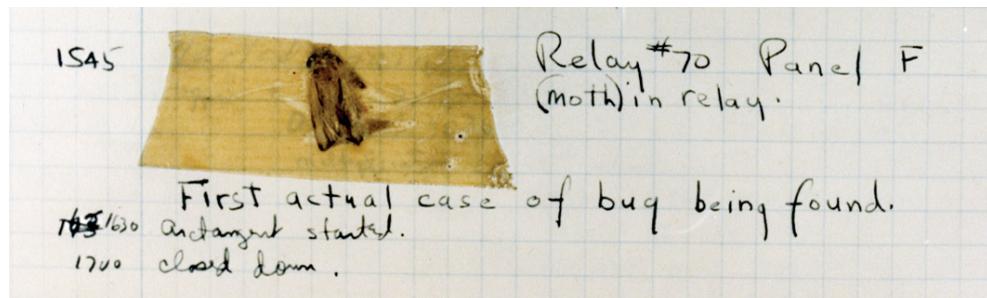



Figura 10.1: Ocorrência do mau funcionamento da máquina Mark II com a mariposa colada

Os técnicos colaram a mariposa no livro de ocorrências, informando que ela era a causa do mau funcionamento da máquina. Grace Hopper então adicionou a frase, embaixo da mariposa: "**Primeiro caso de um bug realmente encontrado**".

Chrome Dev Tools

Neste capítulo vamos usar a ferramenta de debug do Chrome, conhecida como

Developer Tools — e carinhosamente chamada de **Dev Tools**. Não iremos entrar no detalhe de cada *feature* do Dev Tools, mas sim apresentar algumas situações e dicas que — provavelmente — irão melhorar sua rotina de desenvolvedor web.

10.1 Console

"Já é difícil o suficiente encontrar um erro em seu código quando você está procurando por ele; e é ainda mais difícil quando você assume que não existem erros em seu código."
– Steve McConnell

No Console, temos duas diferentes — e complementares — APIs para debugar e melhor saber o que está acontecendo em nossa aplicação. São elas:

- **Console API**: usada principalmente para "logar" informações sobre a aplicação.
- **Command Line API**: um *shell* onde podemos interagir com nossa aplicação, para depurar de forma melhor um bug.

Podemos usar a Console API diretamente em nosso código para imprimir — "logar" — o que está acontecendo em nossa aplicação em tempo de execução. Já a Command Line API está disponível **apenas no contexto do Console** do Developer Tools.

Neste capítulo, vamos tentar fugir do básico, como `console.log` e falar sobre alguns métodos e técnicas menos conhecidas porém muito úteis no nosso dia a dia de caçadores de bugs.

Medindo tempo — `console.time`

O método `console.time` permite — de forma fácil — saber quando tempo determinada operação demorou. Chame o método `console.time` passando uma *string* como argumento para inicializar a contagem do tempo. Para finalizar, chame o método `console.timeEnd` passando a mesma *string* que foi passada anteriormente.

A seguir, um exemplo de medição de tempo em uma requisição XHR (Ajax) à API do GitHub.

```
// medindo tempo de uma requisição XHR (Ajax)
console.time('GitHub API demorou');
$.get('https://api.github.com/search/repositories?q=tetris')
  .done(function() {
    console.timeEnd('GitHub API demorou');
  });
```

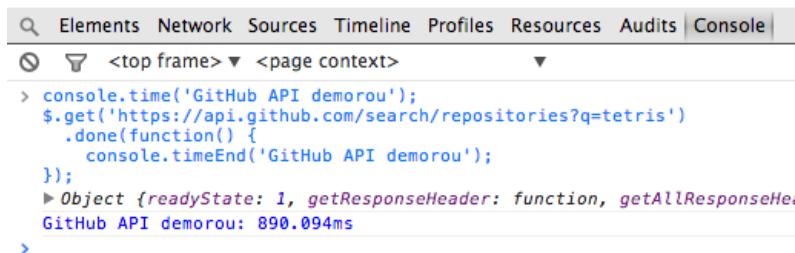



Figura 10.2: Saída do console.time

Logando dados tabulares — `console.table`

Com o `console.table`, podemos imprimir no Console dados como uma tabela. Ótimo para logar uma resposta de uma API, como no exemplo anterior. Poderíamos imprimir de uma forma mais fácil de visualizar a resposta da API da seguinte forma:

```
$.get("https://api.github.com/search/repositories?q=tetris")
  .done(function(response) {
    console.table(response.items);
  });
```

É possível escolher quais campos serão mostrados. Passe um `array` como segundo argumento informando quais chaves dos objetos serão mostradas, como no exemplo a seguir.

```
$.get("https://api.github.com/search/repositories?q=tetris")
  .done(function(response) {
    console.table(response.items, ["full_name", "description"]);
  });
```

Figura 10.3: Saída no console do método `console.table`

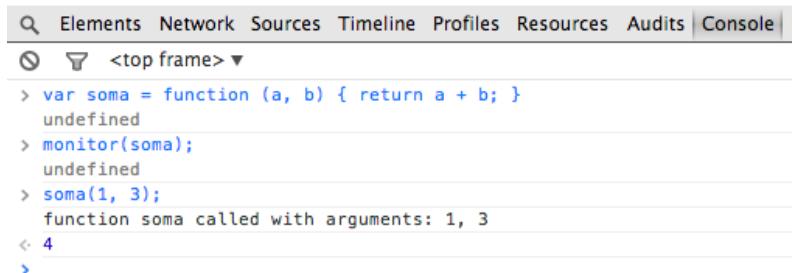
Imprimindo a stack trace — `console.trace`

Isso é muito útil se usado no lugar correto. Ele imprime a *stack trace* do ponto onde foi chamado, incluindo links para as linhas dos devidos arquivos JavaScript.

```
console.trace();
```

Listando todos os *event listeners* — `getEventListeners`

Esta função retorna todos os `event listeners` do objeto passado como parâmetro. É uma mão na roda na hora de debugar código.

```
// Retorna uma lista com todos os event listeners do elemento document
getEventListeners(document);
```

Listando todas as regras CSS de um elmento — `getMatchedCssRules`

Retorna uma lista de todas as regras CSS que estão sendo aplicadas no objeto passado como parâmetro.

```
// Retorna uma lista com todas as regras CSS aplicadas ao elemento <body>
getMatchedCSSRules(document.querySelector("body"));
```

Monitorar chamadas a uma função — `monitor`

Monitora todas as chamadas à função passada como parâmetro. Toda vez que a função `monitor(fn)` for chamada, esta chamada é logada no Console mostrando o nome da função, parâmetros e seus valores.

Figura 10.4: Monitorando chamadas a uma função

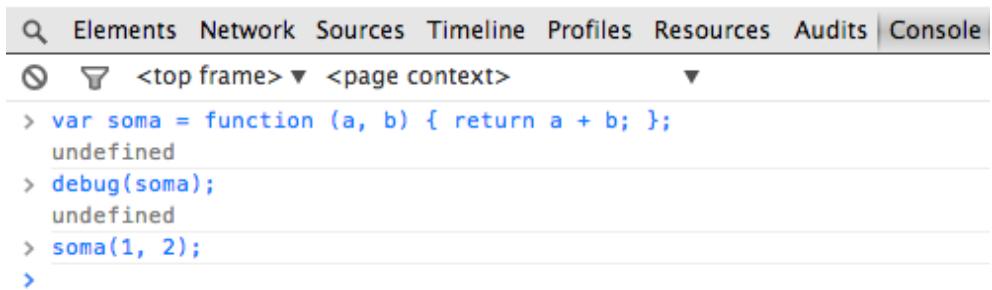
A função `unmonitor` desliga o monitoramento na função passada como parâmetro.

Monitorando eventos — `monitorEvents`

Quando algum dos eventos especificados acontece no objeto passado como parâmetro, o objeto `Event` é logado. Caso não seja especificado nenhum parâmetro, todos os eventos serão escutados.

Figura 10.5: Monitorando todos os eventos no objeto `window`

Para filtrar quais eventos serão monitorados, passe como segundo parâmetro um `array` com uma lista dos mesmos.

```
// Monitorando apenas os eventos click, resize e scroll
monitorEvents(window, ["click", "resize", "scroll"]);
```

Também é possível especificar **tipos** de eventos, que funcionam como grupos predefinidos de eventos. Os tipos disponíveis são:

- **mouse**: `mousedown`, `mouseup`, `click`, `dblclick`, `mousemove`, `mouseover`, `mouseout`, `mousewheel`;
- **key**: `keydown`, `keyup`, `keypress`, `textInput`;
- **touch**: `touchstart`, `touchmove`, `touchend`, `touchcancel`;
- **control**: `resize`, `scroll`, `zoom`, `focus`, `blur`, `select`, `change`, `submit`, `reset`.

A função `unmonitorEvents` desliga o monitoramento na função passada como parâmetro.

Breakpoint em uma função — `debug`

Adiciona um `breakpoint` na primeira linha da função passada como parâmetro. Com esse método, fica mais fácil debugar uma função sobre a qual não sabemos em que arquivo ela está implementada.

```
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame> ▼ <page context> ▼
> var soma = function (a, b) { return a + b; };
  undefined
> debug(soma);
  undefined
> soma(1, 2);
>
```

Figura 10.6: Adicionando breakpoint em uma função pelo método `debug`

10.2 Utilizando breakpoints

"Se debugar é o processo de remoção de bugs do software, então programar é o processo de introduzi-los."
– Edsger Dijkstra

Um *breakpoint* é uma pausa intencional na execução do código de sua aplicação para melhor ajudar no processo de debug. De uma forma mais abrangente, um

breakpoint é uma forma de adquirir conhecimento sobre seu programa em tempo de execução. Neste capítulo, vamos aprender como usar estes breakpoints, como funcionam e por que eles são nossos melhores rifles na temporada de caça aos bugs.

O código que servirá de exemplo

Vamos usar o seguinte trecho de código (problemático) para ilustrar o uso de breakpoints. Uma função que adiciona um valor a um *array* apenas se este valor já não existir previamente.

```
// Adiciona o primeiro argumento ao array passado como segundo argumento
// apenas se o primeiro não existir no segundo.
function pushIfNotExists(val, array) {
  var i;

  for (i = 0; i < array.length; i++) {
    if (cmp(array[i], val)) {
      return false;
    }
  }

  array.push(val);

  return true;
}
```

Essa função recebe dois argumentos. O primeiro argumento é o valor a ser inserido no array e o segundo argumento é o array em que iremos inserir o primeiro argumento, apenas se este não existir previamente no array.

Esta mesma função declarada aqui utiliza uma outra função para fazer comparação entre valores, a função `cmp`. Ela tem a seguinte implementação:

```
// Retorna `true` caso o primeiro argumento seja igual ao segundo.
// `false` caso contrário.
function cmp(a, b) {
  if (a == b) {
    return true;
  } else {
    return false;
  }
}
```

O código que irá usar a função `pushIfNotExists` é o seguinte:

```
// Definimos o valor inicial do array
var arr = [1, "", 3];

// Tentamos inserir os números de 0 a 4 na variável `arr`.
// Como 1 e 3 já existem, 0, 2 e 4 deveriam ser inseridos
for (var i = 0; i < 5; i++) {
  pushIfNotExists(i, arr);
}

console.log(arr.sort());
```

O valor esperado da função seria `["", 0, 1, 2, 3, 4]`, porém se você executar o código acima, verá que, na verdade, a variável `arr` ficou com o valor final de `["", 1, 2, 3, 4]`, com o número 0 faltando. Tinhamos uma expectativa em relação ao comportamento deste algoritmo, porém esta expectativa não foi satisfeita. **Temos um bug**.

Keyword debugger

Vamos configurar um breakpoint para entender o que está acontecendo em nossa aplicação em tempo de execução. Dentro do `for` que tenta inserir novos valores à variável `arr`, vamos inserir a *keyword* `debugger`, como a seguir.

```
for (var i = arr.length; i >= 0; i--) {
  debugger; // adicione esta linha ao código
  pushIfNotExists(i, arr);
}
```

Rode novamente seu código no Chrome e você verá que ele irá parar a execução exatamente onde declaramos a *keyword* `debugger`. Declarar um `debugger` no código é um dos meios de inserir um breakpoint.

Agora começa o processo de debug. Vamos nos aprofundar neste código (em tempo de execução) e entender o que está acontecendo.

Controlando o fluxo de execução

Neste exato momento, caso você esteja rodando seu código no Chrome, sua tela deve desta forma:

Figura 10.7: Execução do código parada devido ao breakpoint

A execução do JavaScript está parada e nós podemos ver o estado de todas as variáveis. Ponha seu mouse sobre as variáveis para ver seu valor atual.

O que queremos saber é por que o valor `0` não está sendo inserido na variável `arr`. Então estamos procurando pelo momento em que a variável `i` do *loop* `for` possui o valor `0`. Continue a execução do código apertando no botão destacado na imagem a seguir.

Figura 10.8: Pressione este botão para continuar a execução do *script*

Como a *keyword* `debugger` está dentro de um loop, o código irá parar novamente. Verifique o valor da variável `i` pondo o mouse em cima dela. Ela deve estar com o valor `4` agora, já que ela começou com `5`.

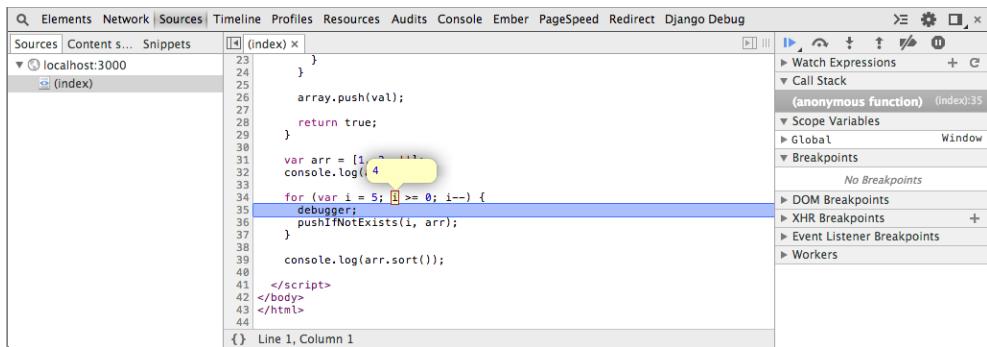



Figura 10.9: Ponha o mouse em cima de qualquer variável para saber seu valor

Continue com a execução do *script* até que o valor da variável `i` chegue a `0`, pois sabemos que é nesta iteração do loop que o bug ocorre. Uma vez com a variável `i` com o valor `0`, vamos debugar passo a passo o que está acontecendo.

O *script* está agora parado na linha onde inserimos a *keyword* `debugger`. Para executar o *script* linha a linha, clique no botão que fica ao lado direito do botão de continuar a execução, como destacado na imagem a seguir.

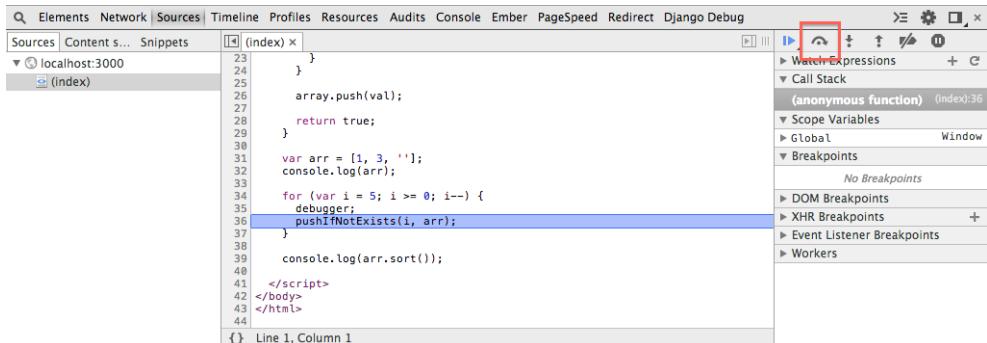



Figura 10.10: Clique aqui para executar o código passo a passo

Agora o código está parado na chamada da função `pushIfNotExists`. Como estamos na última linha e iteração do loop, o problema só pode estar dentro da função `pushIfNotExists`. Nós podemos "entrar" na chamada de uma função para entender como esta função irá se comportar. E é exatamente isso que queremos. Clique no botão destacado na imagem a seguir para "entrar" na função.

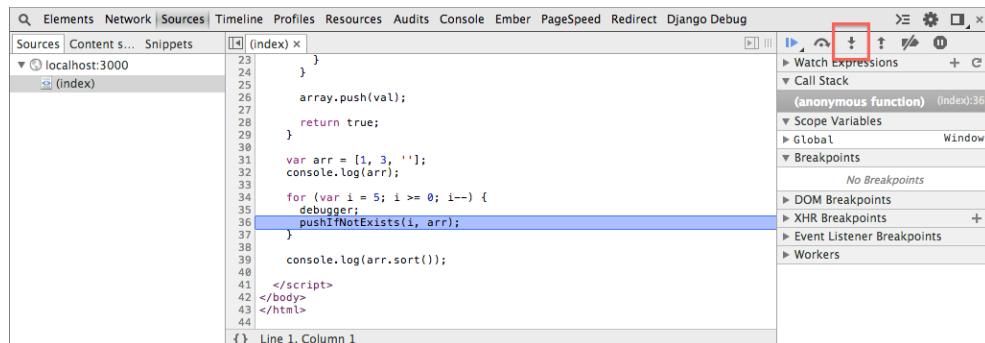



Figura 10.11: Clique aqui para "entrar" na função

Vendo o código da função `pushIfNotExists`, vemos que antes de inserir o valor no array, verificamos se este valor prestes a ser inserido não é igual a nenhum dos valores já existente. Nós verificamos isso comparando todos os valores do array com o valor a ser inserido.

Aperte o botão para irmos à próxima chamada de função, o botão que tem o desenho de uma seta passando por cima de um ponto. Vá até o loop `for` e nele vamos debugar cada iteração.

Continue apertando o mesmo botão para que o código seja executado passo a passo, até que chegue na linha `return null`, que é quando a função `cmp` diz que já existe um valor destes no nosso código. Quando chegar nesta linha, verifique quais valores estão sendo passados para a função `cmp`. Isso pode ser feito pondo o mouse em cima das variáveis, ou escrevendo no Console as variáveis. Como já sabemos como verificar o valor das variáveis através do *mouse over*, vamos agora usar o Console. Você pode ir para a aba **Console** ou apertar no botão mostrado na figura abaixo para chamar o *drawer* e ficar com a aba *Sources* e com o Console visíveis ao mesmo tempo.

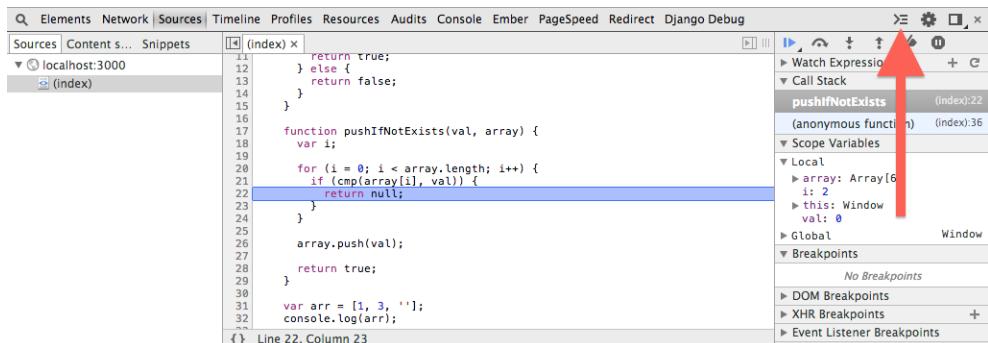



Figura 10.12: Chame o drawer para ter um Console aberto ao mesmo que o Sources

Agora verifique o valor das variáveis passadas para a função `cmp`. Uma vez com a execução do código parada no nosso breakpoint, o contexto na aba **Console** reflete o contexto do ponto em que nosso código está parado. Então, podemos verificar o valor das variáveis no Console e, até, modificá-las.

Para verificar o valor das variáveis, apenas digite seus nomes no Console e aperte `enter`, como na imagem abaixo.

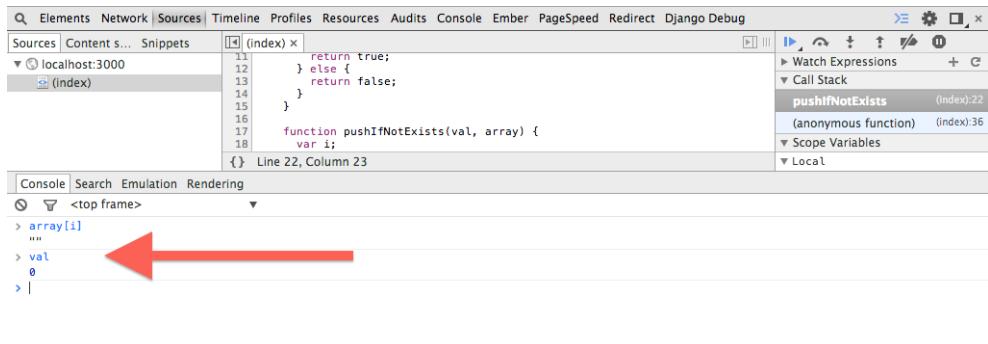
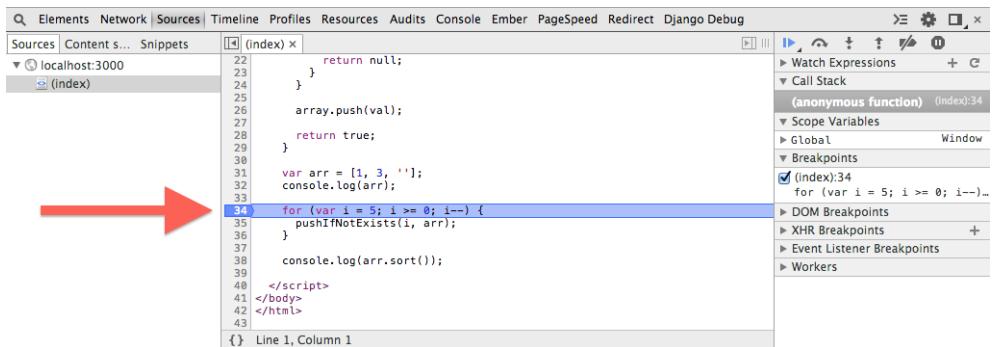



Figura 10.13: Verificando valores das variáveis passadas para a função cmp

Como podemos ver, a função `cmp` retorna `true` quando compara `0` com uma *string* vazia. Por que isso? Vamos averiguar novamente a sua implementação.

Para ir direto na implementação de uma função, digite `command` + `shift` + `O` no OS X ou `Ctrl` + `Shift` + `O` no Windows e Linux. Irá aparecer um *dialog*, digite `cmp` e depois `Enter`. Estamos cada vez mais próximos do problema.

Como podemos ver, a função `cmp` utiliza o operador `==` para comparar se dois valores são iguais. Mas o JavaScript é uma linguagem fracamente tipada, ou seja, quando vamos comparar valores de tipos diferentes (aqui estamos comparando uma *string* com um número), é feito, de forma implícita, uma coerção. Apesar de uma variável ser uma *string* e a outra um número, para o JavaScript, o número `0` e uma *string* vazia são equivalentes (não idênticas, mas equivalentes).

Comparador == e === no JavaScript

O operador `==`, chamado de comparador abstrato, primeiro transforma os dois operandos para um tipo comum (todos para números, `string` etc.) e só então reliza a comparação. O operador `===`, chamado de comparador `strict`, só retorna `true` caso os dois operandos sejam do mesmo tipo e tenham valores iguais.

Como boa prática, utilize sempre o operador `===` e você não terá surpresas desagradáveis.

Agora sabemos onde está o nosso bug. Estávamos comparando apenas a equivalência entre dois valores e não se eram de fato iguais. Para que nosso código agora funcione sem bugs, basta trocar o operador `==` pelo `===`. Desta forma, estamos comparando valor e tipo entre as variáveis.

Breakpoint visual

Podemos repetir todo o processo anterior inserindo breakpoints de forma visual sem a necessidade de escrever a *keyword* `debugger` dentro do código. Na aba *Sources*, basta clicar sobre o número da linha que queremos adicionar um breakpoint.

Figura 10.14: Setando *breakpoints* de forma visual

Como exercício, tente refazer todos os passos que realizamos anteriormente setando o breakpoint de forma visual pelo próprio Dev Tools.

Breakpoint condicional

Breakpoints condicionais, como o próprio nome sugere, são breakpoints que só irão ser disparados caso uma determinada condição seja satisfeita. Nós poderíamos ter simplificado o processo de debug do exemplo anterior caso tivessemos usado um breakpoint condicional.

Nós já sabíamos que o erro estava ao tentar inserir o número `0`. Então, podemos pedir que a execução do *script* seja interrompida apenas nessa condição. Dentro da função `pushIfNotExists`, vamos setar um breakpoint condicional quando o argumento `val` for igual a `0`. Clique com o botão secundário no número da linha no começo do loop `for` e escolha a opção **Add conditional breakpoint...** Dentro da condição escreva `val === 0`.

Quando esta expressão retornar `true`, o *script* irá parar sua execução e podemos debugar nossa aplicação da mesma forma como fizemos antes, só que, dessa vez, com menos passos.

Figura 10.15: Breakpoint condicional

Breakpoints em *event listeners*

É possível também inserir um breakpoint em *event listeners*. Ainda na aba *Sources*, na coluna da direita, expanda a seção **Event Listener Breakpoints** e escolha o evento que deseja debugar.

Figura 10.16: Breakpoint em event listeners

Será adicionado um breakpoint em todos os *listeners* dos eventos escolhidos à direita e, uma vez que o evento for disparado, o *script* irá parar de ser executado na primeira linha da função de *callback* do *listener*.

Debugando eventos de mutação do DOM

Também é possível inserir breakpoints caso uma dada subárvore do DOM seja

modificada. Em outras palavras, caso uma `div` seja inserida, um `p` removido, uma classe adicionada a algum elemento etc.

Para inserir um breakpoint, vá para a aba **Elements**, clique com o botão direito em cima do nó (subárvore) que deseja debugar e escolha o tipo de evento que quer que ocorra uma pausa na execução do *script* caso seja disparado.

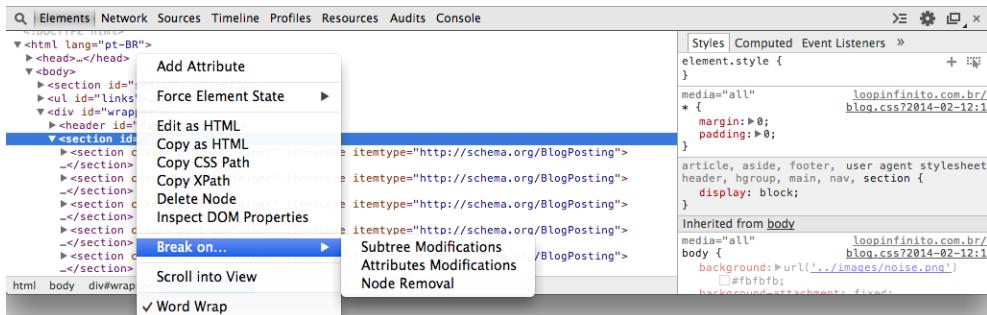



Figura 10.17: Breakpoint em *Mutation events* do DOM

Existem 3 tipos de eventos de mutação do DOM que podem ser escutados para adicionar breakpoints. São eles:

- **Modificação na subárvore**: quando algum nó é adicionado abaixo do nó que está sendo inspecionado.
- **Modificação de atributos**: quando algum atributo do nó inspecionado é modificado como, por exemplo, quando uma classe é adicionada ou removida.
- **Remoção de nó**: quando algum nó da subárvore é removido.

Uma vez que sejam inseridos vários breakpoints em diferentes nós, fica difícil saber onde estão todos para, quando não for mais necessário, removê-los. Na aba **Sources**, na coluna à direita, existe um lista com todos os breakpoints do DOM.

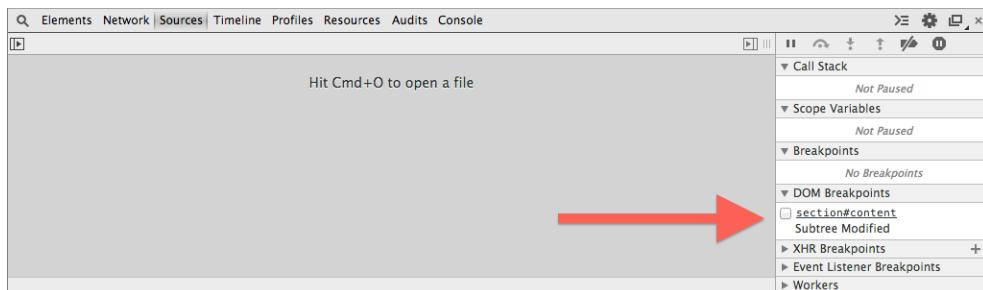



Figura 10.18: Lista de todos os DOM breakpoints

Breakpoints em chamadas XHR (Ajax)

Para debugar código que faz chamadas XHR (Ajax) a uma determinada URL, vá na aba **Sources**, na coluna da direita, no agrupador com o nome de **XHR Breakpoints** e aperte no botão de "+".

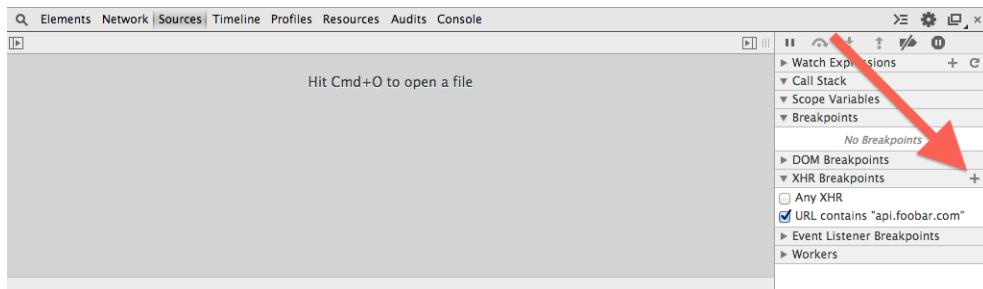



Figura 10.19: Breakpoint em chamadas XHR (Ajax)

Ele vai perguntar a URL (ou parte dela) que deve causar o breakpoint. É só inserir e teremos um breakpoint disparando quando aquela URL for chamada via Ajax.

10.3 Emulando dispositivos móveis

"A ferramenta de depuração mais eficaz ainda é o pensamento cuidadoso, juntamente com instruções de impressão criteriosamente colocadas."
– Brian Kernighan

Debugar uma aplicação web em um dispositivo móvel pode ser uma tortura,

tanto pela falta de ferramentas como pelas peculiaridades de cada dispositivo. Quando falamos de desenvolver para *mobile*, nada substitui o teste no dispositivo real. Mas, neste capítulo, vamos aprender algumas ferramentas que podem acelerar o desenvolvimento para esta plataforma.

Habilitando o painel de emulação de dispositivos

O primeiro passo para brincar com estas novas ferramentas é habilitar o painel de Emulação. Para isso, clique no ícone de *Settings*, no canto direito superior do Dev Tools.

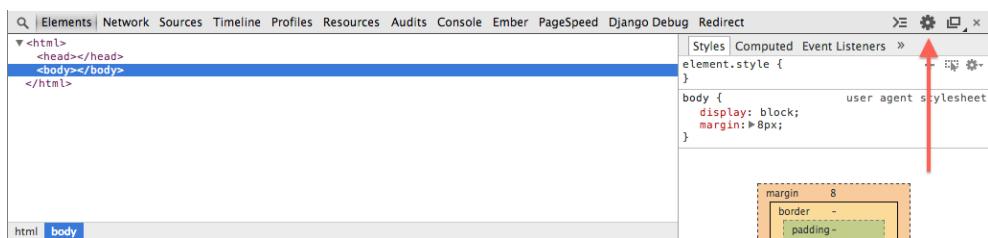



Figura 10.20: Abrindo as preferências do Dev Tools

Agora habilite o *checkbox* **Show 'Emulation' view in console drawer**.

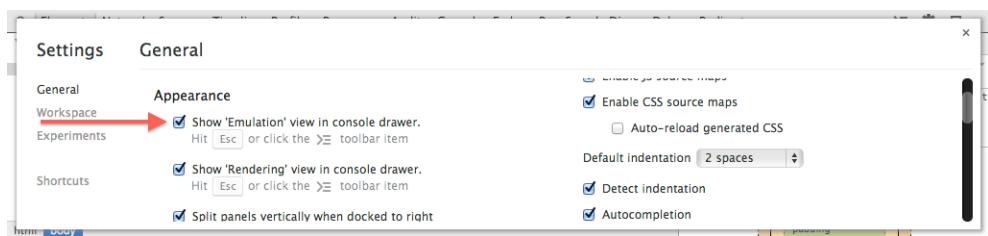



Figura 10.21: Habilitando a aba Emulation no drawer

Depois, basta chamar o próprio *drawer* e clicar na nova aba **Emulation**.

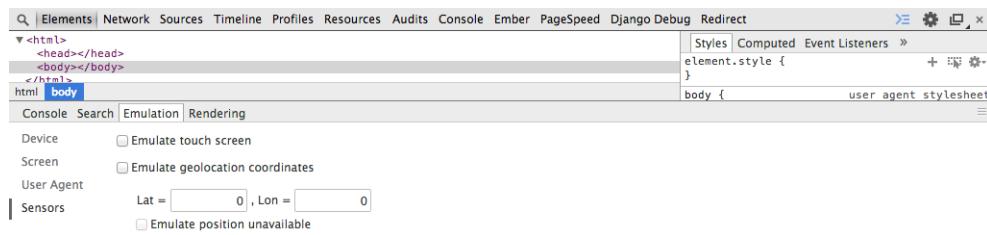



Figura 10.22: Opções da aba Emulation

O *drawer* também pode ser chamado pressionando a tecla `Esc`.

Emulando dispositivos

Normalmente, começamos a desenvolver páginas ou aplicações para a web testando em um browser desktop e, só depois, vamos testar em um dispositivo móvel para, então, corrigir todos os bugs que encontramos no dispositivo real.

No Dev Tools é possível melhorar esse fluxo emulando algumas características de dispositivos móveis e testar alguns casos que só aconteceriam no dispositivo real, como um breakpoint de uma *media query* no CSS. As características que podem ser emuladas são:

- **User agent** — uma *string* disponível em `navigator.userAgent` que também é passada no cabeçalho de uma requisição HTTP. Algumas aplicações verificam se o dispositivo é um dispositivo móvel ou não por essa *string*.

- **Screen resolution** — emula as dimensões reais (largura e altura) de um dispositivo.

- **Device Pixel Ratio** — permite que seja emulado um dispositivo com tela "retina" a partir de um dispositivo "não-retina". O que quer dizer que algumas *media queries*, tais como `@media (min-resolution: 2dppx)` podem ser testadas.

- **Emulate viewport** — aplica um *zoom out* ao *viewport* físico padrão do dispositivo.

- **Text autosizing** — emula algumas mudanças na renderização da fonte efetuadas por alguns dispositivos móveis, a fim de melhorar a legibilidade.

- **Android font metrics** — o Android aumenta de forma artificial o tamanho da fonte de acordo com algumas configurações de sistema e tamanho de tela. Esta opção é habilitada por padrão apenas quando se está emulando dispositivos Android.

Por padrão, o Dev Tools já vem pré-carregado com configurações dos dispositivos móveis mais populares atualmente. Escolhendo um dispositivo da lista, ele irá popular as outras abas (**User Agent**, **Screen**, entre outras) com as configurações específicas do dispositivo escolhido.

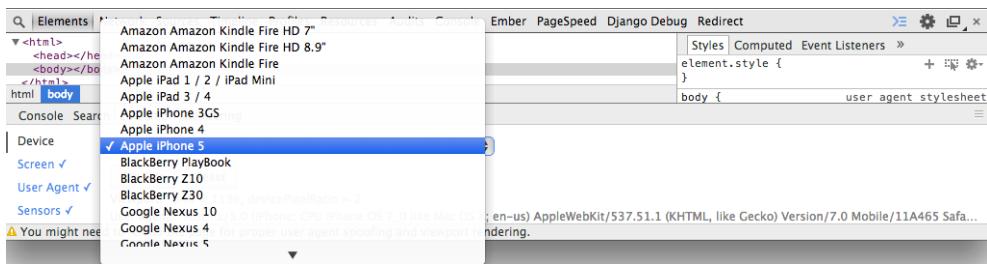



Figura 10.23: Selecionando dispositivo a ser emulado

Caso deseje emular um dispotivo não listado, é possível editar os campos individualmente com as especificações do aparelho desejado.

Emulando eventos touch

Ter que subir uma modificação para um servidor e dar *refresh* na página do seu dispositivo móvel pode ser um saco na hora de desenvolver algo relacionado com *touch events*, umas vez que a maioria dos desktops não possuem esse tipo de *input*.

Para eventos do tipo *single-touch* (feitos apenas com um dedo), é possível simulá-los pelo Dev Tools. No painel **Emulation**, vá para a aba **Sensors** e habilite o *checkbox* **Emulate touch screen**, como na figura a seguir.

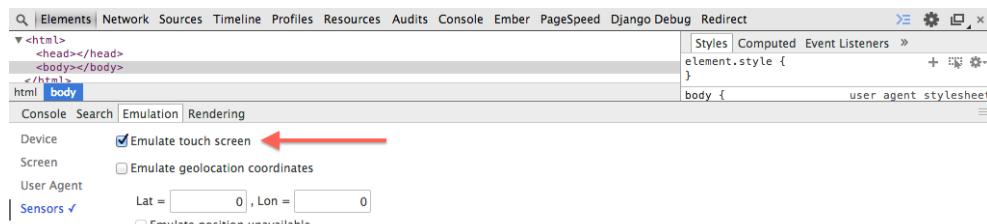



Figura 10.24: Emulando touch events

Seu mouse irá virar uma círculo, simulando a ponta de um dedo. Um clique do mouse irá disparar os eventos touch. Para simular um *pinch*, aperte `Shift` e, com o botão do mouse pressionado, mova o ponteiro.

Spoofing de User Agent

O *User-agent* é uma *string* que identifica o browser — agente. Esta *string* é passada no cabeçalho de toda requisição ao servidor e algumas aplicações fazem uso dessa forma de identificar o navegador para servir um conteúdo otimizado.

No Dev Tools, é possível alterar esta *string* fazendo com que o navegador se "disfarce" de um outro. Com o *drawer* aberto, vá no painel **Emulation**, na aba **User Agent** e escolha o *User Agent* de um dispositivo ou entre com um específico, como na figura a seguir.

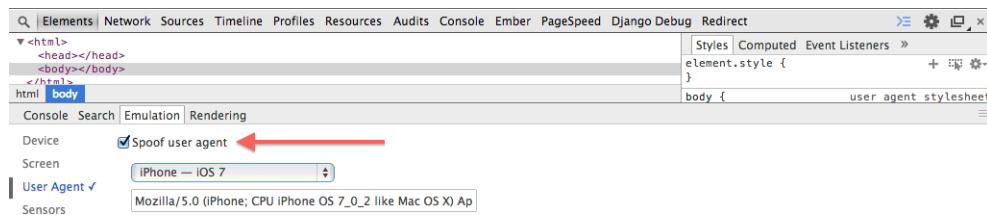



Figura 10.25: Spoofing de Useragent

Um dos websites que faz tal tipo de detecção é o google.com. Faça um *spoofing* de *User Agent* do iPhone e tente visitar o google.com. Um nova página será renderizada.

Sobreescrevendo geolocalização

Imagine que um usuário de sua aplicação reporte um bug ao tentar procurar por restaurantes na cidade do Rio de Janeiro. Você mora em São Paulo e sua aplicação recebe esses dados pela API de Geolocalização do dispositivo. Como simular o cenário do usuário que reportou o bug e verificar quais as repostas da aplicação no contexto da cidade do Rio?

Essa informação também pode ser sobreescrita para fins de debugar código. No *drawer*, vá para o painel **Emulation**, na aba **Sensors** e habilite o *checkbox* **Emulate geolocation coordinates** informando a latitude e longitude que você deseja passar emular, como na figura a seguir.

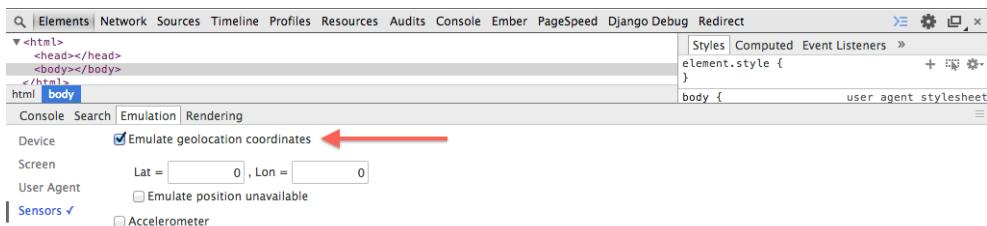



Figura 10.26: Sobreescrevendo geolocalização

Emulando acelerômetro

Computadores desktop e a maior parte dos notebooks não possuem acelerômetro. A ausência desse sensor pode tornar impossível o teste de um código que acesse os dados de aceleração do dispositivo. Mas com o Dev Tools é possível emulá-los (ou sobrescrever seu valor). Para isso, vá ao *drawer*, depois na aba *Emulation* e clique no *checkbox* **Accelerometer**. Você pode mexer a representação 3D do dispositivo para emular uma determinada posição ou por valores diretamente em cada eixo.

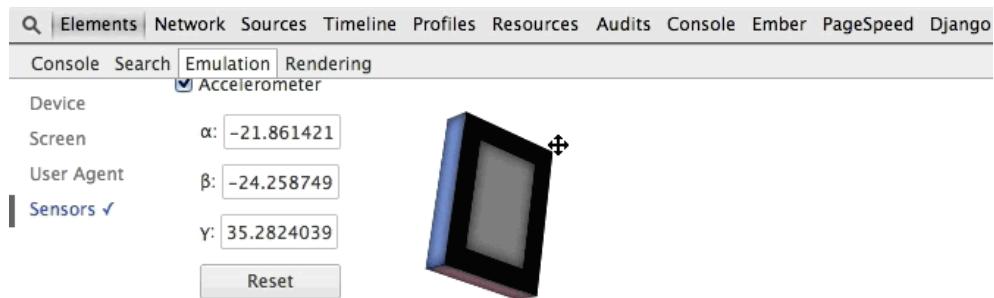



Figura 10.27: Emulando acelerômetro

10.4 Debug remoto

Emular dispositivos em nosso ambiente de desenvolvimento é um salvador de vidas. Mas já imaginou se pudéssemos debugar no desktop um website rodando em nosso celular? Seria o melhor dos dois mundos. A velocidade do desenvolvimento no desktop com a fidelidade da renderização de um dispositivo real, e não emulado.

Pois nossos sonhos viraram realidade. Hoje, com um dispositivo rodando o Android 4.0+ e Chrome, é possível conectar um Dev Tools rodando em seu desktop ao browser rodando no celular.

Habilitando o modo *developer* em seu Android

Para usar o debug remoto, primeiro se deve habilitar a opção *USB debugging* no seu Android.

No Android 4.2+ o menu Developer options está desabilitado por padrão. Para habilitá-lo vá em **Settings**, depois em **About phone** e clique 7 vezes em **Build number** (é sério...). Volte para a tela anterior, clique em **Developer options** e habilite o **USB Debugging**. No Android 4.0 e 4.1, a opção **USB Debugging** está em **Settings > Developer options**.

Figura 10.28: Configuração da depuração USB no Android

Debugando no desktop

Hoje o Chrome desktop suporta de forma nativa o debug em dispositivos conectados por USB, sem mais o antigo estresse de ter que baixar, instalar e configurar o ADB (Android Debug Bridge).

O próximo passo agora é habilitar o debug remoto no Chrome desktop. Visite a página chrome://inspect/#devices e habilite o *checkbox* **Discover USB devices**, como na figura a seguir.

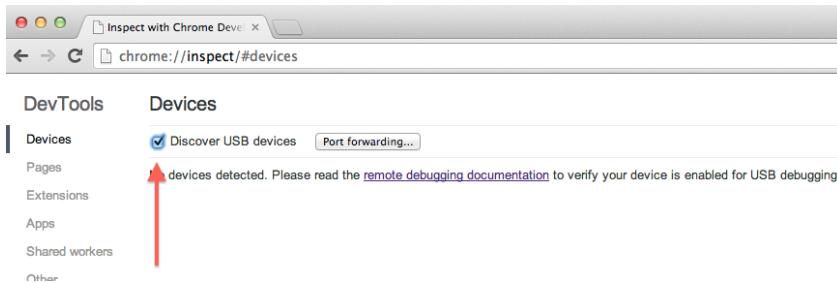



Figura 10.29: Descoberta de dispositivos conectados por USB

Agora conecte seu dispositivo Android ao computador pelo cabo USB. Uma mensagem em seu Android deve aparecer perguntando se deseja habilitar o debugging pela USB. Diga que sim.

Figura 10.30: Configuração da depuração USB no Android

Após isso, seu aparelho será listado no endereço chrome://inspect/#devices. Neste endereço, são listados todos os aparelhos conectados e suas respectivas tabs.

Podemos ter vários aparelhos conectados a um mesmo computador, assim como várias *WebViews*. Escolha qual tab quer debugar e clique em **Inspect**.

Uma nova janela irá aparecer com o querido Dev Tools e um "espelho" da tela do aparelho que estamos debugando. Este espelho do aparelho também é interativo. Podemos, por exemplo, rolar a página, clicar e usar nosso teclado desktop para entrarmos com dados de uma forma mais rápida.

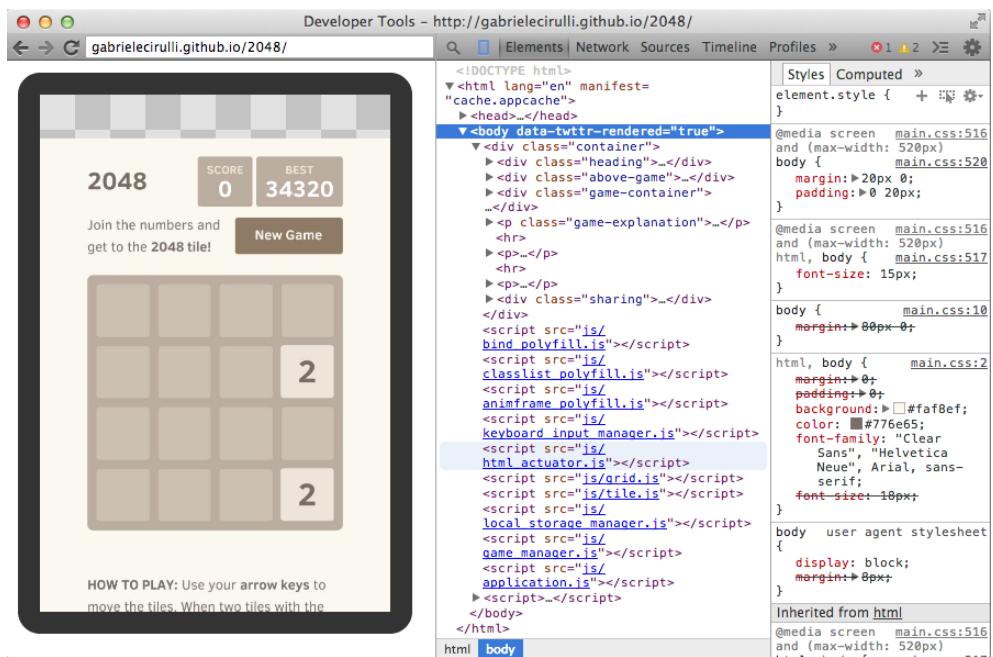



Figura 10.31: Página web rodando em um dispositivo móvel e sendo debugada no Dev Tools no desktop

Agora já podemos começar a debugar nossa aplicação. Como momento **WTF?!**, vá na aba **Inspect** do Dev Tools e inspecione um elemento. Repare que, de acordo com que o mouse passa sobre o HTML dos elementos no Dev Tools, eles ficam marcados com um *overlay* azul **em seu Android**. Assim como acontece quando inspecionamos um elemento em uma página sendo renderizada localmente.

A partir desse momento, você tem um Dev Tools completo, acessando uma página sendo renderizada em um dispositivo real, e não emulado.

Redirecionamento de porta

Normalmente, temos um servidor local rodando em nossa máquina durante o desenvolvimento. Para fazer com que o seu celular faça uma requisição ao servidor que está rodando na máquina de desenvolvimento basta digitar o IP da máquina de desenvolvimento e a porta onde o servidor está escutando.

Isso caso ambos aparelhos estejam na mesma rede, o que nem sempre é verdade. Como, por exemplo, em redes corporativas que geralmente são cheias de restrições.

Para facilitar esse *workflow*, o Dev Tools suporta o redirecionamento de portas. Com isso, podemos fazer com que o dispositivo conectado por USB acesse uma porta no computador ao qual está conectado, mesmo estando em redes diferentes.

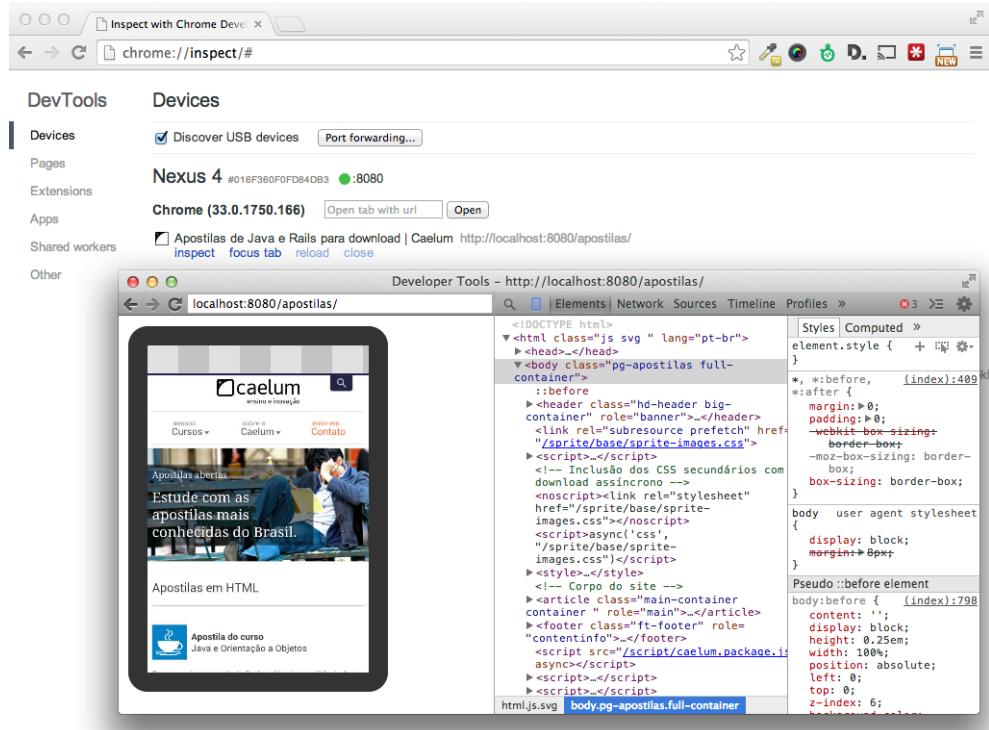



Figura 10.32: Site da Caelum rodando no servidor de desenvolvimento sendo acessado por um dispositivo móvel através de redirecionamento de porta

Para habilitar o redirecionamento, siga os passos:

1) Abra o endereço chrome://inspect/#devices;

2) Clique no botão **Port forwarding**;

3) No campo **Port**, digite a porta em que o dispositivo deverá escutar;

4) No campo **IP address and port** digite o IP mais porta em que seu servidor está rodando;

5) Habilite o *checkbox* **Enable port forwarding** antes de apertar **Done**.

10.5 Dicas e truques

"Antes de um software ser reutilizável, ele precisa primeiro ser usável."
– Ralph Johnson

O Dev Tools é repleto de funções e ferramentas. Mas algumas delas não são tão óbvias de serem achadas. Neste capítulo vamos focar justamente nessas funcionalidades: nas menos comuns. Mesmo você já sendo um rato do Dev Tools, muito provavelmente irá aprender alguns novo truques depois de ler esse capítulo.

Procurando texto entre todos os arquivos carregados

Um dos atalhos que mais uso. Ele procura por um texto ou expressão regular entre todos os arquivos carregados pelo site. Inclusive — e por isso ele é tão útil — entre arquivos que foram carregados em tempo de execução.

Caso esteja no OS X, utilize as teclas `command` + `option` + `F` para disparar o painel de busca. No Windows e Linux, o mesmo é disparado com as teclas `Ctrl` + `Shift` + `F`.

Figura 10.33: Procurando texto entre todos os arquivos carregados

Forçando um estado em um elemento

Já tentou inspecionar um elemento no seu estado de `hover` e, quando você move seu mouse para o Dev Tools, o elemento perde o estado de `hover` e você não sabe mais o que fazer ou quem xingar?

No Chrome, é possível forçar um estado em um elemento, seja ele `active`, `focus`, `hover` ou `visited`. Na aba Elements, inspecione normalmente o elemento e, com ele selecioanado, clique no ícone que fica à direita na parte de cima do Dev Tools e escolha o estado que deseja forçar no elemento selecionado.

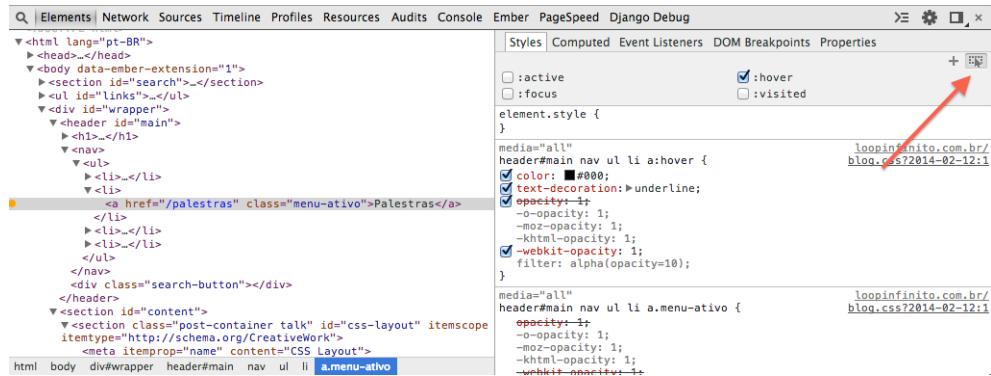



Figura 10.34: Forçando um estado em um elemento

Contador de FPS

Útil para visualizar, em tempo real, o *frame rate* de sua aplicação. Quando ativo, irá aparecer uma caixa preta no topo à direita com detalhes sobre a quantidade de *frames* que estão sendo renderizados por segundo. Para uma experiência ótima, tente sempre ficar acima de 60 FPS.

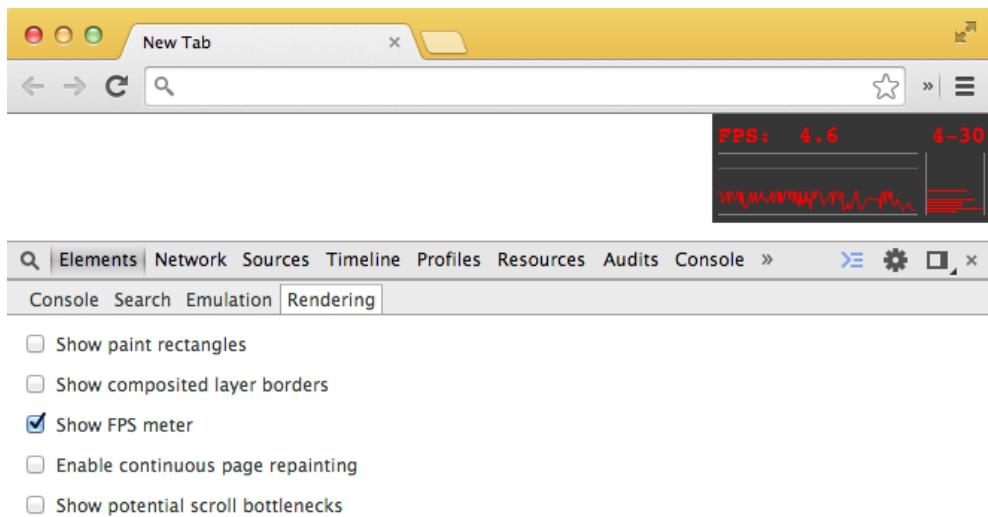



Figura 10.35: Contador de FPS

A informação é útil para você monitorar a performance de renderização do site. Sites com baixo FPS parecem ter animações menos suaves e performance pior.

Replay de chamadas XHR

Já pensou ter que refazer sempre aquele fluxo enorme na sua aplicação só para disparar novamente uma certa requisição Ajax (XHR)? Com esse atalho, você pode dar um replay em qualquer chamada Ajax.

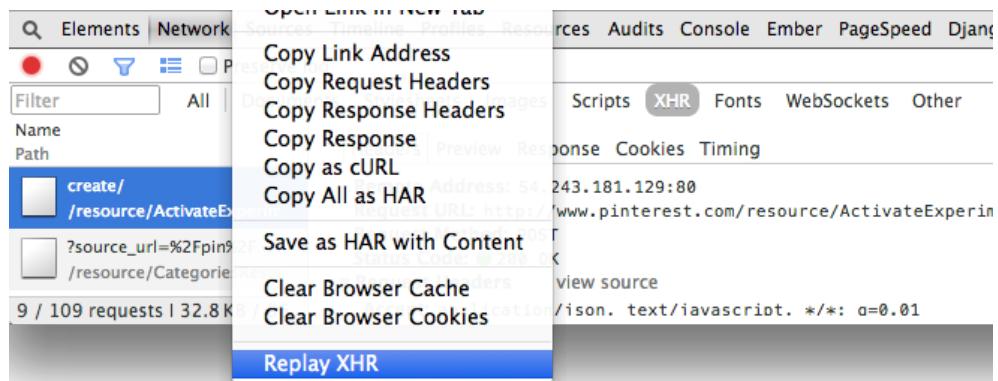



Figura 10.36: Replay de chamadas XHR

Copiando requisições como comando cURL

Praticamente uma derivação do replay de chamadas Ajax. Com a diferença de que, aqui, é gerado um comando cURL da requisição feita com todos os parâmetros e cabeçalhos. Cole no seu terminal e aperte `enter` para disparar uma requisição idêntica à que foi disparada pelo seu navegador.

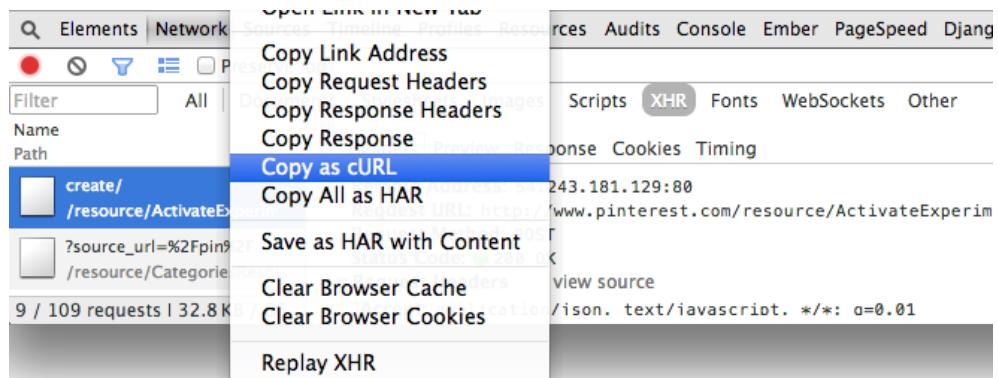



Figura 10.37: Copiando requisições como comando cURL

Troque rapidamente entre abas

Para navegar de forma rápida entre as abas do Dev Tools, utilize `Cmd` + `[` e `Cmd` + `]` (ou `Ctrl` + `[` e `Ctrl` + `]` no Windows e Linux).

Preservar o *log* do console durante navegação

Sabe quando você quer debugar algo que acontece um pouco antes de um *redirect* mas este mesmo redirect apaga o log do console? Para resolver esse problema, clique com o botão direito no console e escolha a opção **Preserve log upon navigation**. Seus logs não serão mais zerados ao visitar uma outra URL.

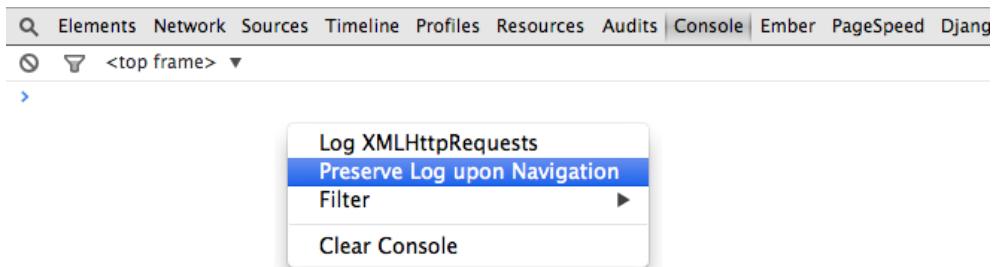



Figura 10.38: Preservar o log do console durante navegação

Limpando o histórico do console

Bastante útil quando se começa a usar com mais frequência a aba Console. `Cmd + K` no OS X e `Ctrl + L` no Windows e Linux para limpar o console. Também funciona com o método `console.clear()`.

Compartilhar e analisar uma Timeline gravada por outro

Imagine que você está com um bug na renderização da sua aplicação que a está deixando com uma experiência ruim, como se o *scroll* estivesse pesado. Então você grava uma sessão com a ferramente **Timeline** e, mesmo assim, não consegue identificar o erro e gostaria muito de enviar esta mesma sessão recém-gravada para um amigo desenvolvedor lhe ajudar no processo de debug.

Hoje isso é possível. Tanto exportar como carregar uma sessão exportada por um terceiro. Uma vez na aba **Timeline**, clique com o botão secundário do mouse dentro da aba e escolha a opção **Save Timeline Data...** para salvar a sessão atual e **Load Timeline data...** para carregar uma sessão.

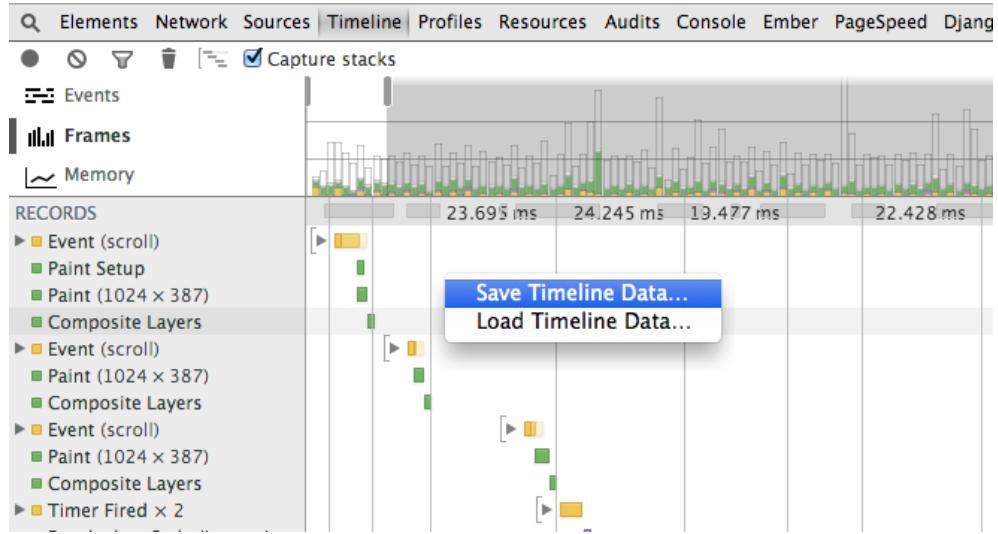



Figura 10.39: Compartilhar e analisar uma Timeline gravada por outro

10.6 Extensões

"Computadores são bons em seguir ordens, mas não em ler mentes."
– Donald Knuth

Existem várias extensões para o Dev Tools que podem melhorar ainda mais o processo de debug de alguns *workflows* específicos. A seguir, apresento algumas ótimas extensões para os mais populares *frameworks*, para otimização de um modo geral e para melhor debugar arquivos no ambiente de produção.

AngularJS Batarang

Extende o Dev Tools com ferramentas específicas para o framework AngularJS.

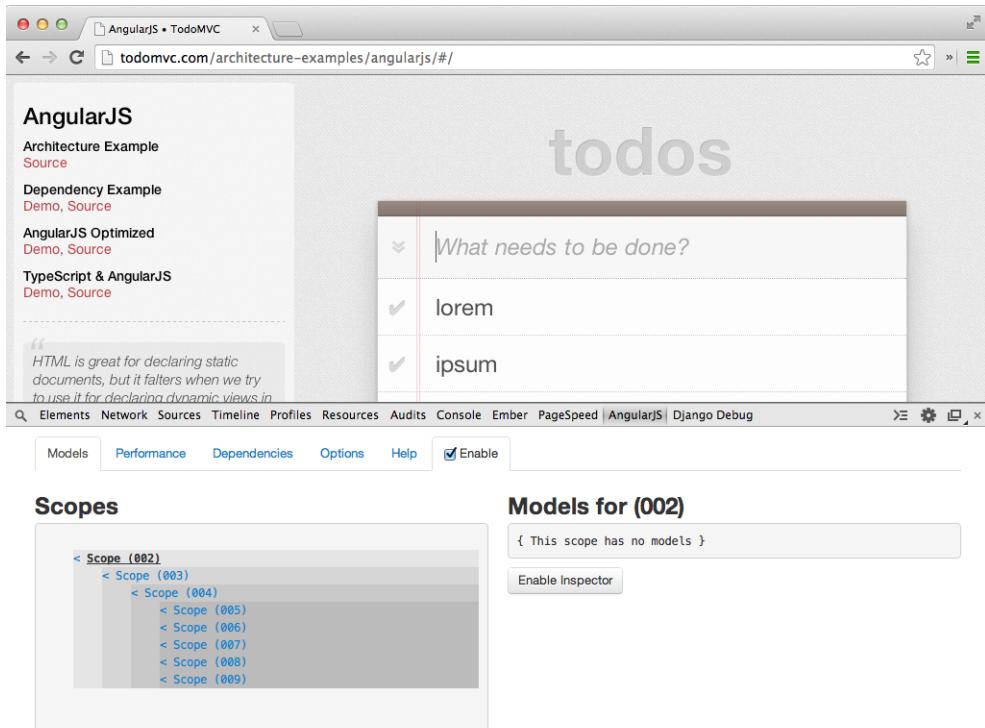



Figura 10.40: AngularJS Batarang

Ember Inspector

Uma ótima extensão para quem trabalha com o framework Ember. Com ele é possível ver todas as rotas sendo usadas, assim como um *overlay* com quais *templates*, *controllers* e *models* estão sendo usados em tempo de execução.

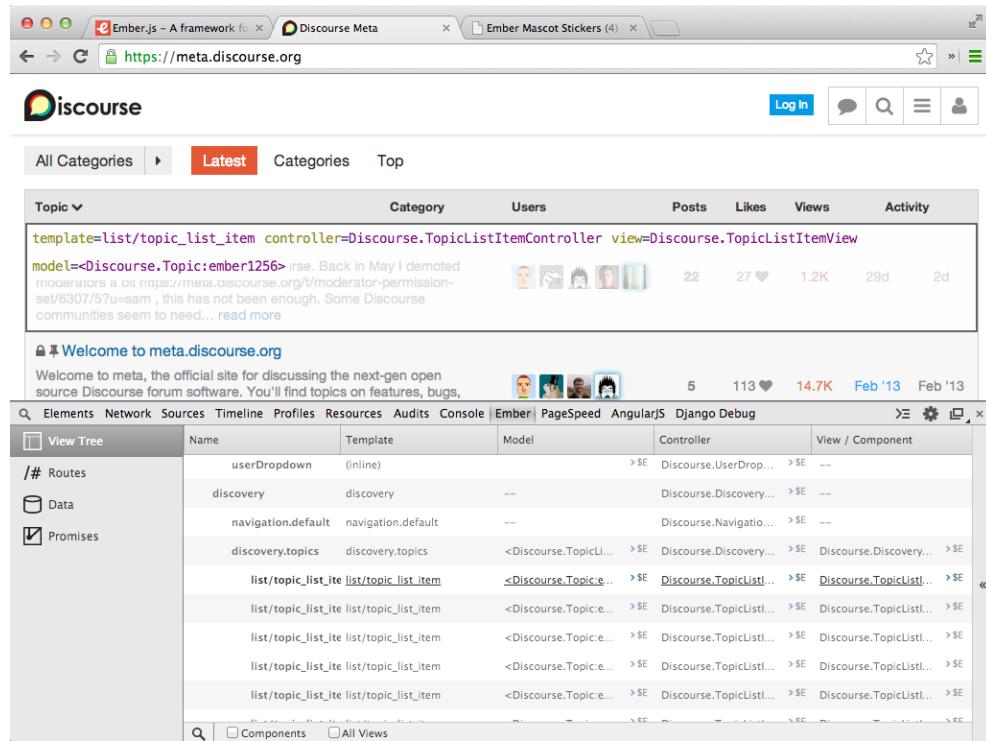



Figura 10.41: Ember Inspector

Backbone Debugger

Mostra, em tempo real, todas as *views*, *models*, *collections* e *routers* usadas por sua aplicação que utiliza o framework Backbone.

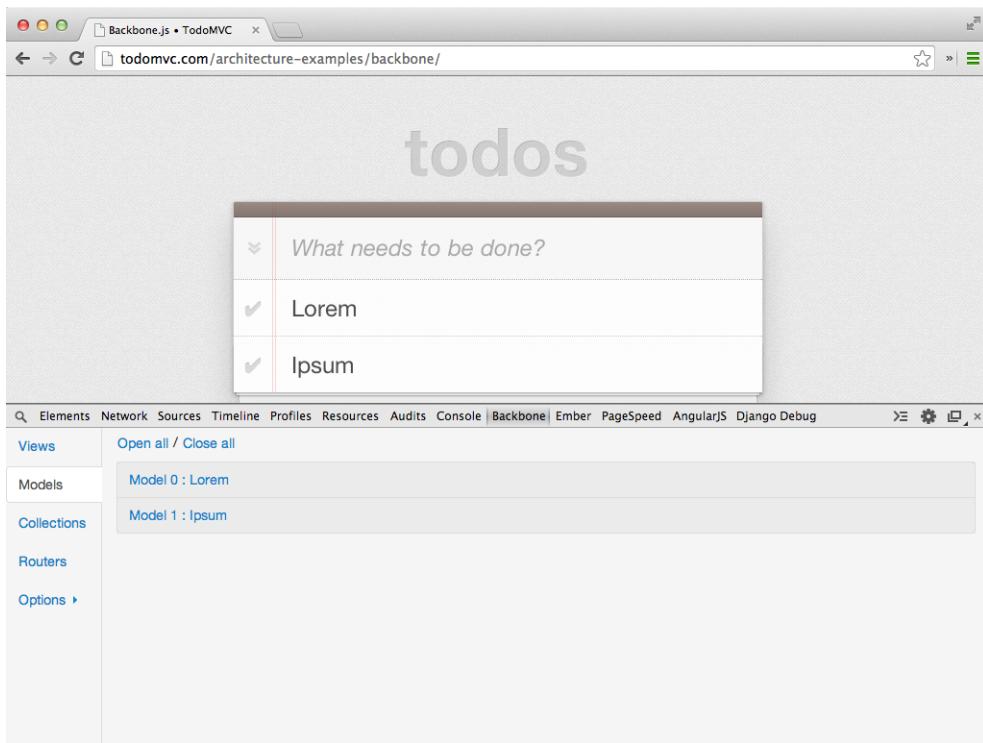



Figura 10.42: Backbone Debugger

Grunt Devtools

Rode todas suas tarefas do Grunt sem sair do browser.

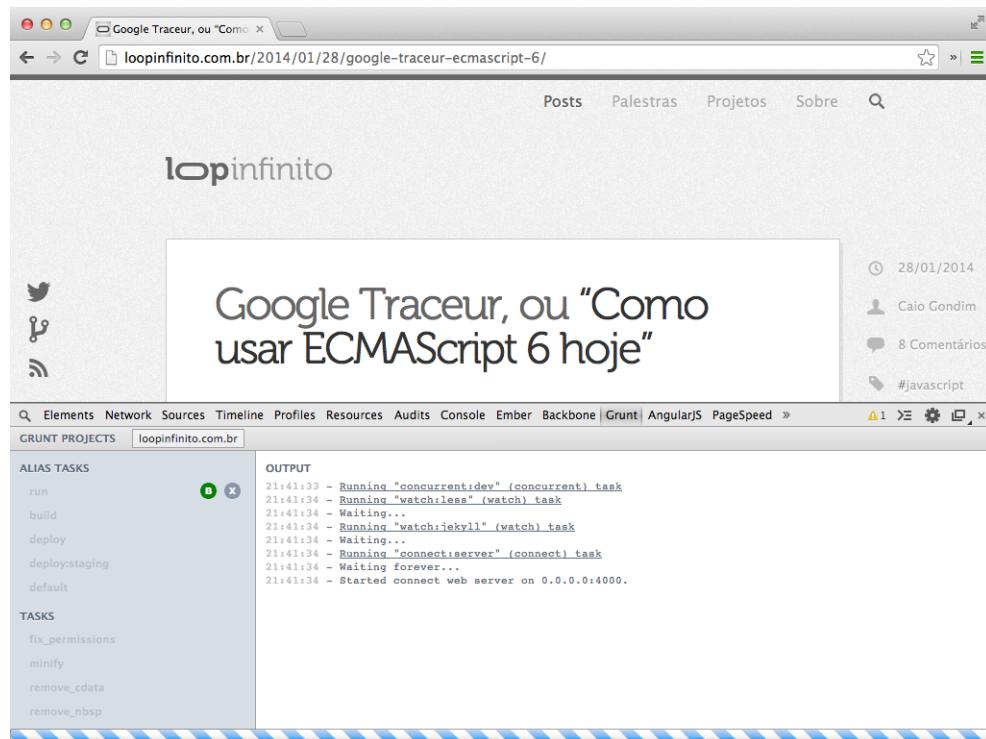



Figura 10.43: Grunt Devtools

Rails Panel

É para quem trabalha com Ruby on Rails. Mostra informações de sua app tais como tempo gasto em cada requisição e tempo total de cada *query* ao banco de dados.

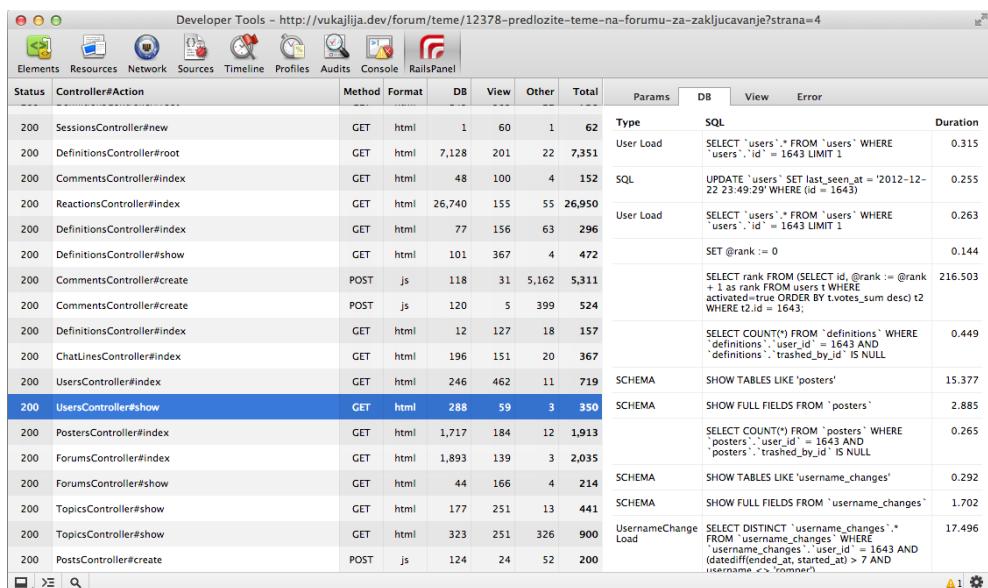



Figura 10.44: Rails Panel

Devtools Redirect

O Devtools Redirect permite que uma requisição feita pelo browser seja redirecionada para um arquivo local em sua máquina. Baixe o arquivo que está em produção, faça suas alterações localmente, e redirecione a requisição para este arquivo em sua máquina. Agora toda vez que o arquivo for carregado, a sua cópia local que será carregada. Ótimo para fazer testes no ambiente de produção que não irá impactar nos usuários finais.

PageSpeed Insights

Esta extensão feita pelo Google nos dá dicas — mais aprofundadas que as da aba *Audits* do Dev Tools — sobre como e onde melhorar a velocidade de carregamento de nossa aplicação.

10.7 Conclusão

"Suponho que seja tentador, se a única ferramenta que você tem é um martelo, tratar tudo como se fosse um prego."

– Abraham Maslow, 1966

O Dev Tools é a ferramenta de debug do Chrome e, em minha humilde opinião, hoje a melhor ferramenta para debug de uma aplicação web. Porém, o código de sua aplicação deve rodar não apenas em um browser, mas em quantos for possível. E bugs irão eventualmente aparecer no momento em que você for testar sua aplicação em diferentes navegadores. Logo, é também necessário — pelo menos — conhecer as ferramentas de debug destes outros browsers. O **Web Inspector** do Safari, o **Firebug** e **Developer Tools** do Firefox são também ótimas ferramentas e não tão diferentes assim das que vimos neste capítulo.

Para se aprofundar mais no assunto, aconselho a leitura dos seguintes materiais:

- **Chrome Dev Tools**: https://developers.google.com/chrome-developer-tools/
- **Safari Web Inspector**: http://bit.ly/1d7Livt
- **Firefox Developer Tools**: https://developer.mozilla.org/en/docs/Tools

Porém debugar uma aplicação não se trata apenas de ferramentas. É necessário experiência para saber por onde começar. Apenas encher seu código de breakpoints sem nenhum propósito não vai ajudar a resolver um bug. É preciso **desconfiar** da coisa certa. E este *know-how* só o tempo irá te dar.

Sobre o autor

Caio Gondim é um desenvolvedor full-stack ex-Globo.com, atualmente trabalhando na Booking.com, recém-chegado em Amsterdam. Interessado em viagens, minimalismo, interface homem-máquina e estilo de vida em geral. Também é palestrante e cofundador do blog loopinfinito.com.br, onde escreve sobre as mais novas tecnologias do mundo front-end. Você pode segui-lo no Twitter em @caio_gondim.

Capítulo 11

Testando códigos JavaScript

11.1 Introdução

O JavaScript é, sem sombra de dúvida, a linguagem de programação mais utilizada no desenvolvimento de *sites* e aplicações web. Em fevereiro de 2014, no Github, dos 50 repositórios com maior número de stars, 25 eram focados em JavaScript. A possibilidade de usá-lo para manipular o DOM, fazer requisições assíncronas e escrever códigos executáveis no lado do cliente fez com que os programadores mudassem a mentalidade em relação ao desenvolvimento para a web.

Com o avanço das APIs disponibilizadas pelos navegadores e o aumento da compatibilidade entre eles, a linguagem criada pelo Brendan Eich em 10 dias possibilitou aos desenvolvedores a criação de interfaces ricas, complexas, e altamente interativas. Em 2004, o Google lançou o Gmail — um marco no uso de tecnologias web —, e graças ao surgimento de frameworks como o jQuery, smartphones com browsers modernos e ao uso cada vez maior em servidores, o JavaScript conquistou a posição de linguagem de uso mais ubíquo.

Mas com o crescimento de seu uso, veio o crescimento da complexidade do código escrito e do risco de quebrar o funcionamento de uma aplicação a cada linha alterada. Como implementar uma solução com um bom design e garantir que alterações em um trecho não influenciem (ou quebrem) outras partes?

No mundo do desenvolvimento de software, esse sempre foi o **papel dos testes**. Neste artigo, vamos ver como aplicá-los ao JavaScript, garantindo uma aplicação resiliente, de fácil manutenção, e fácil de ser modificada e adaptada.

11.2 Os benefícios de testar uma aplicação

Imagine a seguinte situação: você precisa implementar um formulário de dados de cartão de crédito para um e-commerce. Ao submetê-lo, uma validação no lado do cliente deve verificar se:

- o nome foi preenchido;
- o número foi preenchido;
- o número é válido para o cartão da bandeira A;
- a data de validade é válida;
- o código de segurança foi preenchido;
- o código de segurança é válido.

As especificações parecem ser simples o suficiente para a validação ser implementada sem qualquer preocupação com testes. Sem perder tempo, você escreve uma função que realiza todas as verificações. Após testar extensivamente no navegador, você considera que está tudo bem e manda o código para produção.

Imagine que, semanas depois, o sistema de pagamento passe a suportar os cartões da bandeira B. Como a validação suporta apenas a bandeira A, você deve alterar o código para suportar ambas. Porém, dependendo da maneira como a implementação foi feita, a adição da bandeira B pode potencialmente afetar o funcionamento de outras partes do formulário.

Como garantir que tudo continua funcionando corretamente? E se, no futuro, forem adicionadas mais bandeiras de cartão de crédito? E se elas influenciarem como o código de segurança é verificado? E se a bandeira A passar a suportar um novo formato de números?

Em um e-commerce, a parte da finalização da compra é uma das mais cruciais para o negócio. Um pequeno erro pode quebrar o funcionamento da página e causar uma perda substancial de dinheiro — tanto em vendas, quanto na fidelização do cliente. Infelizmente, independente do tipo da aplicação, bugs fazem parte do processo de desenvolvimento. E é a maneira como testamos as funcionalidades que define a quantidade e a gravidade deles na hora de o usuário final interagir com o site.

As diferentes camadas de teste

Existem diferentes formas de se testar uma aplicação e cada uma cobre diferentes partes e camadas. Não existe uma forma única ou ideal de aplicá-las e cada projeto/time pode necessitar diferentes métodos e combinações.

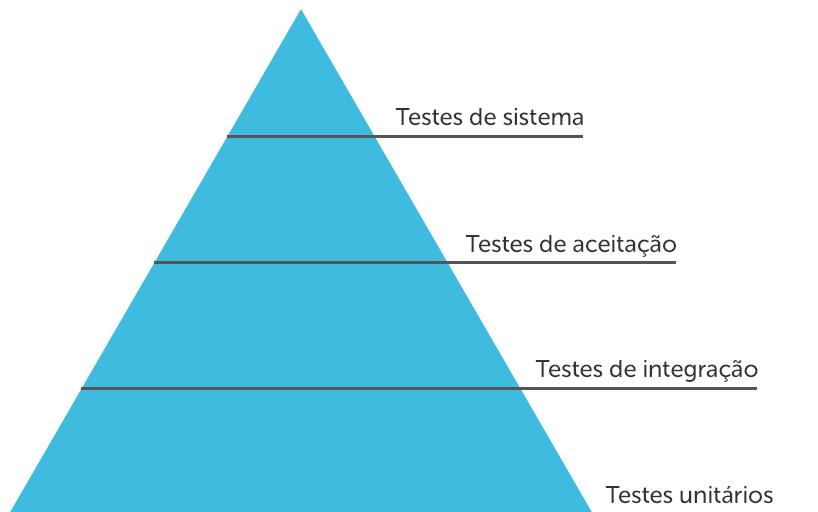



Figura 11.1: As diferentes camadas de teste

- **Testes unitários:** testam as menores partes da aplicação — módulos, fragmentos de código, funções — isoladamente.
- **Testes de integração:** testam a integração entre diferentes módulos e partes do sistema. Em aplicações web, podem testar a comunicação entre componentes em uma mesma página, ou uma feature espalhada entre diferentes seções.

- **Testes de aceitação:** usados principalmente em ambientes ágeis, testam os critérios de aceite de uma estória/feature por completo. Através de ferramentas como o Cucumber (http://cukes.info/) , os *stakeholders primários* (como o *Product Owner*) podem ser envolvidos no processo de criação do teste.
- **Testes de sistema:** testam as features mais cruciais para o funcionamento do site, de ponta a ponta. Um exemplo seria, em um e-commerce, o fluxo de buscar por um produto, adicioná-lo ao carrinho, preencher os dados de entrega e pagamento, e finalizar uma compra.

Como estamos focados no JavaScript, falarei apenas sobre os testes unitários.

11.3 Escrevendo os testes unitários

Para escrever testes para o código JavaScript, precisaremos:

- escolher um framework de testes;
- saber o que é e como escrever uma *spec*;
- organizar as specs em grupos;
- executar os testes e analisar os resultados.

Existem diferentes opções de framework e diferentes formas de usá-los. Aqui, usaremos o **Jasmine**, da Pivotal Labs (http://jasmine.github.io/) . Porém, após aprender os principais conceitos sobre como escrever testes, você verá que é simples aplicá-los em diferentes ferramentas.

Montando o ambiente

Vamos utilizar a versão *"Standalone"* do Jasmine: a mais simples, indicada para pequenos projetos ou bibliotecas. Mais para frente, darei algumas dicas de como utilizá-lo em aplicações maiores.

Entre na página do projeto (https://github.com/pivotal/jasmine/tree/master/dist) , baixe e descompacte o zip da versão mais recente. Na versão 2.0.0, essa é a estrutura do diretório:

```
/jasmine-standalone-2.0.0
  /lib
  MIT.LICENSE
  /spec
    PlayerSpec.js
    SpecHelper.js
  SpecRunner.html
  /src
    Player.js
    Song.js
```

No momento, o que nos interessa é o arquivo `SpecRunner.html` e o conteúdo dos diretórios `spec` e `src`.

Abra o arquivo `SpecRunner.html`. Você verá algo assim:

```
Jasmine 2.0.0                                        finished in 0.008s
•••••
5 specs, 0 failures                                  raise exceptions
  Player
    should be able to play a Song
    when song has been paused
      should indicate that the song is currently paused
      should be possible to resume
    tells the current song if the user has made it a favorite
    #resume
      should throw an exception if song is already playing
```

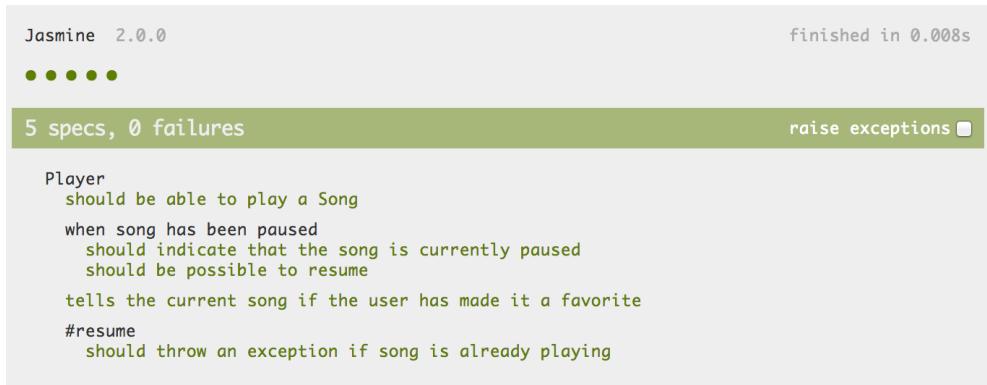



Figura 11.2: SpecRunner.html

Você acaba de rodar testes JavaScript no seu navegador! Claro, são apenas exemplos prontos, mas ver *"5 specs, 0 failures"* já não dá um ânimo?

Agora abra o `SpecRunner.html` no seu editor de código favorito. Você verá uma página HTML incluindo um arquivo CSS e alguns arquivos JavaScript. No final do bloco `<head>`, há este trecho de código:

```
<!-- include source files here... -->
<script src="src/Player.js"></script>
<script src="src/Song.js"></script>
```

```
<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
```

O comentário já dá a dica: você deverá incluir seus módulos/bibliotecas a serem testados e, em seguida, os arquivos contendo os testes.

Apague o conteúdo do arquivo `spec/SpecHelper.js`, apague os arquivos `src/Player.js`, `src/Song.js` e `spec/PlayerSpec.js` e, no HTML, deixe apenas a chamada ao `SpecHelper.js`:

```
<!-- include source files here... -->

<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
```

O `SpecHelper.js` deverá conter quaisquer *helpers* e métodos que você queira criar para auxiliá-lo no futuro. Caso não precise, ele também pode ser removido.

Agora estamos prontos para escrever nossos testes. Como exemplo prático, vamos implementar o formulário de cartão de crédito que citei anteriormente. Mas, primeiramente, é necessário conhecer os conceitos de **spec** e **suite**.

O que é uma spec?

Desde que descompactamos o arquivo zip, você deve ter notado a palavra **spec** em diversos lugares. O que é uma spec, e o que ela tem a ver com testes?

Uma spec é como se fosse a menor "unidade" de um teste. Ela possui uma **descrição** — uma frase curta indicando o que está sendo testado — e uma ou mais **asserções** — uma comparação entre expectativa e resultado que retorna `true` ou `false`. Quando todas as asserções são verdadeiras, a spec **passa** com sucesso. Se alguma delas é falsa, a spec **falha**.

As asserções possuem três partes: a primeira é o valor a ser analisado, chamado de "valor real", a segunda é o valor que queremos, chamado de "valor esperado", e a terceira é a comparação que será feita entre os dois valores.

```
Spec: Deve ter a cor azul
  Asserção: O estilo 'color' de .meu-paragrafo deve ser igual a 'blue'
```

No exemplo, temos:

- Valor real: O estilo 'color' de .meu-paragrafo

- Valor esperado: 'blue'
- Comparação: deve ser igual a

É importante que a descrição da spec não seja muito genérica. No caso do nosso formulário de cartão de crédito, uma spec com a descrição "deve validar corretamente" acabará possuindo muitas asserções. Caso uma delas falhe, a spec inteira falhará e será mais difícil identificar o problema. Quebrar os testes em specs granulares também garantirá um melhor design de código: cada pequena parte do componente será implementada e testada isoladamente.

O que é uma *suite*?

As specs são agrupadas por contextos, chamadas de **suites**. Uma suite pode conter um conjunto de suites — que podem conter outras suites — e uma ou mais specs.

```
Suite: Meu parágrafo
  Suite: Estilos
    Spec: Deve ter a cor azul
    Spec: Deve ter a fonte Helvetica
  Suite: Conteúdo
    Spec: Deve conter o texto 'Hello world!'
```

Geralmente, cada componente é representado por uma suite de mesmo nome, que por sua vez é composta por outras suites representando diferentes partes de sua implementação, cada uma com seu conjunto de specs.

Criando seu primeiro teste

Quando descrevi o exemplo do formulário, listei várias especificações. Cada uma delas é elegível para ser uma spec separada. Algumas podem ser quebradas em até duas ou mais specs distintas. Relembrando, o componente deverá verificar se:

- o nome foi preenchido;
- o número foi preenchido;
- o número é válido para o cartão da bandeira A;
- a data de validade é válida;
- o código de segurança foi preenchido;

- o código de segurança é válido.

Para começar no mundo dos testes, pegarei o primeiro exemplo da lista. Crie o arquivo `validadorCartaoDeCreditoSpec.js` no diretório `spec` e o inclua no `SpecRunner.html`:

```
<!-- include source files here... -->

<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/validadorCartaoDeCreditoSpec.js"></script>
```

No Jasmine, as suites são representadas pela função `describe`. O primeiro argumento é a descrição da suite e o segundo é uma função que representa o bloco de implementação da suite. Esse conceito ficará mais claro à medida que implementarmos as specs.

Abra o arquivo `validadorCartaoDeCreditoSpec.js` e inclua as seguintes linhas:

```
describe("validadorCartaoDeCredito", function () {

});
```

Salve o arquivo e abra o `SpecRunner.html` no navegador. Você acaba de criar e rodar sua primeira suite de testes!

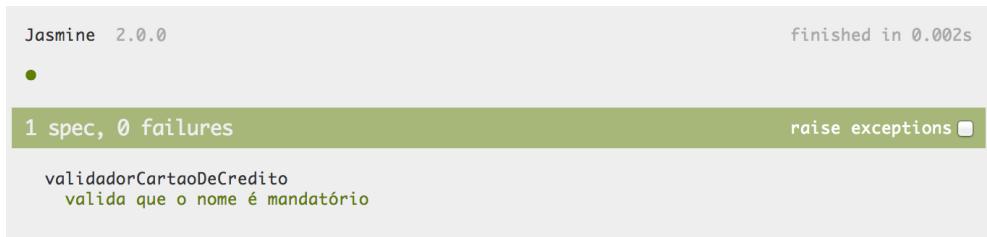



Figura 11.3: Nossa primeira suite de testes

Mas, no momento, ela não está testando nada. Vamos criar nossa primeira spec. Elas são definidas através da função `it` e, assim como a função `describe`, aceitam como argumentos uma descrição e uma função que implementa a spec:

```
describe("validadorCartaoDeCredito", function () {
  it("valida que o nome é mandatório", function () {
```

```
  });
});
```

Ao salvar o arquivo e recarregar a página, você verá que ele listará a spec e indicará que ela passou com sucesso. Como não escrevemos nenhuma asserção, ele assume que tudo deu certo.

Figura 11.4: Nossa primeira spec

Como vimos antes, uma asserção é composta pelo valor real, pelo valor esperado e pela comparação entre ambos. No Jasmine, isso é escrito através da função `expect` e dos **matchers**. Modifique a implementação da seguinte forma:

```
describe("validadorCartaoDeCredito", function () {
  it("valida que o nome é mandatório", function () {
    var nome = "";

    expect(validadorCartaoDeCredito.validaNome(nome)).toEqual(false);
  });
});
```

Opa, tem bastante coisa acontecendo aqui! De onde saiu esse `validaNome`? E esse `toEqual`? Vamos por partes:

- A função `expect` é o que define a asserção no Jasmine. Ela aceita como parâmetro o valor real, ou seja, aquele que queremos garantir que está funcionando corretamente.

- A função `toEqual` é um **matcher** — funções que servem para comparar valores. No caso do Jasmine e de outros frameworks, ele funciona na forma de *chain*, sendo chamado imediatamente depois de outra função. O matcher irá

comparar o valor passado para a função `expect` com o valor passado para ele. Cada um possui uma comparação diferente e facilmente deduzível pelo nome — o `toEqual` compara se os valores são iguais.

- O `validadorCartaoDeCredito` e seu método `validaNome` ainda não existem. É nesse ponto que os testes podem melhorar e até mesmo definir o design da sua implementação. Tendo as descrições das specs definidas, podemos começar a "prever" como nosso código será utilizado, antes mesmo de ele ser implementado. O nome disso é *"Behavior Driven Development"* (Desenvolvimento Movido a Comportamento) — ou BDD — que descreverei com mais detalhes posteriormente.

Ao recarregar o `SpecRunner.html`, você verá que finalmente temos um erro: *"ReferenceError: validadorCartaoDeCredito is not defined"*.

Figura 11.5: Erro: o validadorCartaoDeCredito ainda não foi definido

Como nem implementamos o `validadorCartaoDeCredito`, esse era o resultado esperado. Vamos fazê-lo: crie o arquivo `validadorCartaoDeCredito.js` no diretório `src` e inclua-o no `SpecRunner.html`:

```
<!-- include source files here... -->
<script src="src/validadorCartaoDeCredito.js"></script>

<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/validadorCartaoDeCreditoSpec.js"></script>
```

Abra o arquivo e defina o `validadorCartaoDeCredito`:

```
var validadorCartaoDeCredito = {

};
```

Recarregue o `SpecRunner.html` no navegador. Você verá que a mensagem de erro mudou para *"TypeError: Object #<Object> has no method 'validaNome'"*.

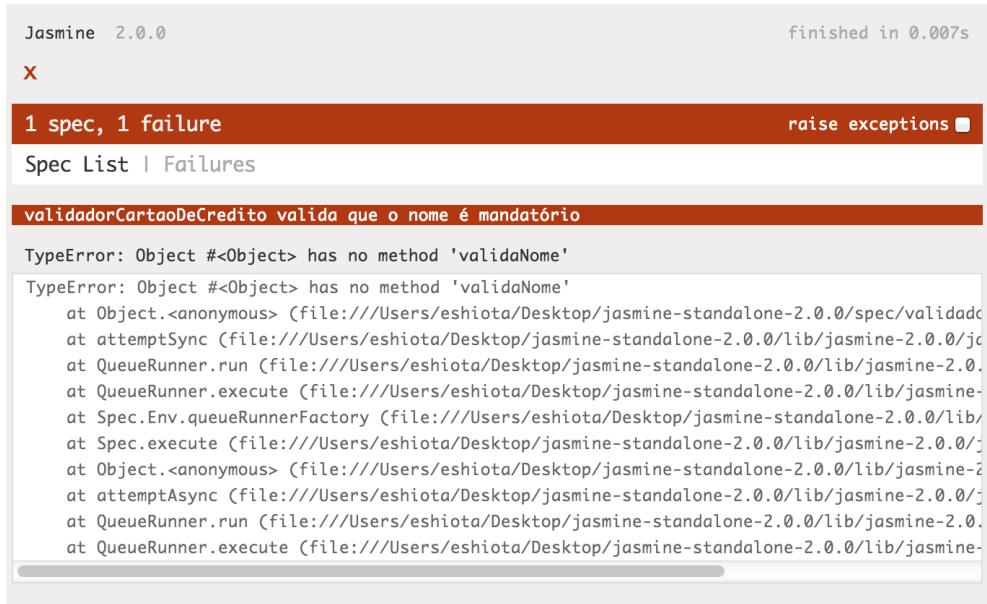



Figura 11.6: Erro: o validadorCartaoDeCredito não possue o método validaNome

Os erros indicam justamente o nosso próximo passo: criamos o `validadorCartaoDeCredito`, agora precisamos criar o método `validaNome`:

```
var validadorCartaoDeCredito = {
  validaNome : function () {
```

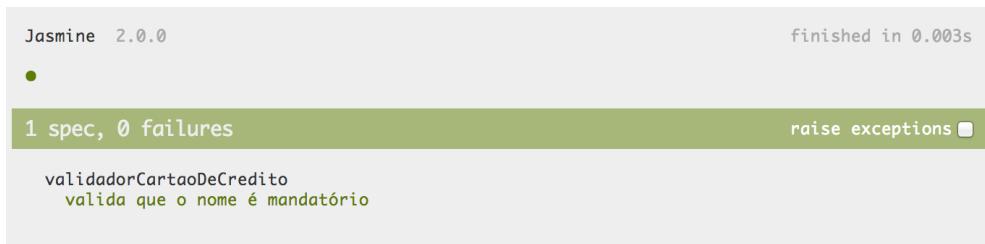
```
  }
};
```

Agora a mensagem mudará para *"Expected undefined to equal false"*. Como `validadorCartaoDeCredito.validaNome()` não retorna nada (ou seja, retorna `undefined`), o teste falha.

Figura 11.7: Erro: undefined deveria ser false

A intenção não é fazer o teste passar? E se fizermos o seguinte?

```
var validadorCartaoDeCredito = {
  validaNome : function () {
    return false;
  }
};
```

```
Jasmine 2.0.0                                        finished in 0.003s
•
1 spec, 0 failures                                   raise exceptions
  validadorCartaoDeCredito
    valida que o nome é mandatório
```

Figura 11.8: O teste passou. Mas a implementação está correta?

Embora a implementação esteja completamente errada, o teste passou, pois `false === false` é verdadeiro. Criamos um teste inútil? Testar é perda de tempo?

Para evitar isso, basta começarmos a ser mais criteriosos e criar mais specs. Nossa suite pode ser melhorada da seguinte maneira:

```
describe("validadorCartaoDeCredito", function () {

  describe("ao validar o nome", function () {

    it("returna true se o nome não estiver em branco", function () {
      var nome = "Shiota";

      expect(validadorCartaoDeCredito.validaNome(nome)).toEqual(true);
    });

    it("returna false se o nome estiver em branco", function () {
      var nome = "";

      expect(validadorCartaoDeCredito.validaNome(nome)).toEqual(false);
    });

  });
});
```

Figura 11.9: Erro: agora uma das specs falha

Agora uma das specs irá falhar e seremos obrigados a corrigir nossa implementação:

```
var validadorCartaoDeCredito = {
  validaNome : function (nome) {
    return nome !== "";
  }
};
```

Ao recarregar a página com os testes, todos passarão.

```
Jasmine 2.0.0                                    finished in 0.002s

2 specs, 0 failures                              raise exceptions

  validadorCartaoDeCredito
    ao validar o nome
      returna true se o nome não estiver em branco
      returna false se o nome estiver em branco
```

Figura 11.10: Todas as specs estão verdes!

Agora, imagine que o código está funcionando e foi para o ar, e alguém conseguiu burlar a validação do nome colocando um espaço em branco no campo de texto.

Antes de corrigir o erro desesperadamente, por mais urgente que seja, abra seu arquivo de testes e adicione uma spec para testar esse caso:

```
it("retorna false se o nome é um espaço em branco", function () {
  var nome = " ";

  expect(validadorCartaoDeCredito.validaNome(nome)).toEqual(false);
});
```

Figura 11.11: A nova spec pegou uma falha na implementação

E não é que falhou mesmo? Vamos corrigir no `validadorCartaoDeCredito.js`:

```
var validadorCartaoDeCredito = {
  validaNome : function (nome) {
    return nome.trim() !== "";
  }
};
```

Pronto. Paz restaurada e a spec adicionada garantirá que esse bug nunca mais passará batido.

Desenvolvendo com BDD/TDD

Como vimos anteriormente, é possível escrever os testes antes mesmo de implementar a solução. A vantagem disso é que, tendo em mãos as especificações e o resultado que queremos, podemos de antemão projetar e prever como será o design da implementação.

Existem duas técnicas para isso: o **BDD**, que usamos para escrever nossa spec do validador de cartão de crédito, e o **TDD** — *Test Driven Development*, ou Desenvol-

vimento Movido a Testes. Ambos são muito similares, diferindo apenas na maneira como descrevemos e organizamos as suites e specs.

No **BDD**, os testes definem o **comportamento** do nosso código. O resultado é uma leitura de fácil compreensão e um pouco mais desprendida de conceitos técnicos. O Jasmine tem foco em BDD, e o nome dos seus métodos enfatiza as descrições das suites e specs. No exemplo que desenvolvemos, a execução de uma spec formula uma frase:

[quote Validador de cartão de crédito, ao validar o nome, ele retorna true se o nome não estiver em branco.]

No **TDD**, os testes definem a **estrutura exata** do nosso código. Ao invés do comportamento, descrevemos a implementação. O QUnit, por exemplo, é focado em TDD, e seus métodos refletem isso:

```
module("validadorCartaoDeCredito");

test("#validaNome", function () {
  var validador = validadorCartaoDeCredito;

  equal(validador.validaNome("Shiota"), true, "valida nome");
  equal(validador.validaNome(""), false, "valida em branco");
  equal(validador.validaNome(" "), false, "valida espaço");
});
```

Entre TDD ou BDD, não existe escolha certa ou errada. Contanto que os testes unitários estejam sendo escritos e que isso seja feito antes da implementação, ambos melhorarão a qualidade do seu código e do seu projeto.

11.4 No mundo real

"Legal Shiota, mas como vou usar tudo isso na vida real?"

A maioria dos tutoriais de testes e BDD descrevem situações simples e otimizadas — incluindo os meus exemplos. Mas, no dia a dia, as aplicações são muito maiores e complexas. Como continuar escrevendo testes?

Escrevendo specs pequenas

Não importa se sua aplicação é composta apenas por um validador de cartão de crédito, ou por centenas de componentes em centenas de arquivos: continue escrevendo specs simples, pequenas, e sempre quebre os problemas em partes menores.

Uma aplicação de médio porte, como um e-commerce, pode facilmente ter mais de 1000 specs. Se todas estiverem devidamente separadas em arquivos e suites, e forem pequenas e de fácil compreensão, sua aplicação estará devidamente coberta, resiliente a erros e poderá crescer ou ser modificada sem grandes preocupações.

Lidando com o DOM

Uma das maiores desculpas contra testes unitários em JavaScript é não poder testar código que mexe com o DOM — e, em uma aplicação web, isso representa 95% dos casos. Isso é balela: se os testes rodarem em um navegador que disponibiliza o DOM, a implementação será totalmente testável.

No exemplo do cartão de crédito, poderíamos criar um componente para gerenciar o nó do DOM correspondente ao formulário, e testá-lo:

```
describe("formCartaoDeCredito", function () {
  describe("ao submeter o formulário", function () {
    it("deve validar o nome", function () {
      var form = document.createElement("form")
        , evento = document.createEvent("HTMLEvents")
        , validador = validadorCartaoDeCredito
      ;

      spyOn(validador, "validaNome");

      formCartaoDeCredito.init(form);

      evento.initEvent("submit", true, true);

      form.dispatchEvent(evento);

      expect(validador.validaNome).toHaveBeenCalled();
    });
  });
});
```

Através da API de manipulação do DOM e de técnicas como *injeção de dependência*, é possível testar comportamentos típicos de navegadores sem depender de o formulário estar presente de fato no HTML. Neste exemplo, um elemento `<form>` foi criado do zero, e o evento `submit` foi simulado para testar o comportamento do `formCartaoDeCredito`.

Rodando os testes em linha de comando

Não há problemas em usar o `SpecRunner.html` do Jasmine para testar pequenos códigos e bibliotecas, mas para projetos grandes, é necessário um mecanismo um pouco mais sofisticado do que abrir um arquivo HTML com chamadas para JavaScript. Para facilitar, automatizar, e integrar os testes com servidores de integração contínua (http://blog.caelum.com.br/integracao-continua/) , vamos utilizar uma *Command Line Interface* (CLI) — Interface de Linha de Comando — e rodar as specs direto do terminal, sem precisar de um navegador.

Existem diversas soluções, cada uma atendendo a um conjunto diferente de frameworks e disponibilizando diferentes resultados. Neste artigo, utilizarei o Karma (http://karma-runner.github.io/) , uma ferramenta que possibilita rodar testes de diversos frameworks, em diferentes navegadores — inclundo o PhantomJS, um navegador *headless* (sem interface visual, ideal para testar apenas em linha de comando).

Primeiramente, é necessário possuir o Node.js instalado em sua máquina. Caso não possua, visite o site (http://nodejs.org/) e instale a última versão. Você deverá também instalar o PhantomJS (http://phantomjs.org/) .

O Node.js possui uma ferramenta de gerenciamento de pacotes, o NPM (Node Package Manager), e o Karma pode ser instalado através dela. Abra o terminal e digite na linha de comando:

```
$ npm install -g karma
```

Instale também o `karma-cli`, que disponibilizará o comando `karma` em qualquer diretório:

```
$ npm install -g karma-cli
```

Vamos utilizar os mesmos arquivos que criamos para rodar com o `SpecRunner.html`. Crie um novo diretório (`meuprojeto`, por exemplo) e copie os diretórios `src` e `spec` para lá:

```
/meuprojeto
  /spec
  /src
```

Entre no diretório `meuprojeto`, e digite:

```
$ karma init karma.config.js
```

O Karma fará diversas perguntas e criará um arquivo de configuração `karma.config.js` para você. Siga as instruções e responda de maneira similar ao exemplo a seguir.

```
Which testing framework do you want to use ?
Press tab to list possible options.
Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options.
Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options.
Enter empty string to move to the next question.
> PhantomJS
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
> src/*.js
> spec/*Spec.js
>

Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> yes
```

Uma breve explicação sobre as opções:

- Qual framework usaremos? Além do Jasmine, o Karma suporta Mocha, QUnit, nodeunit e NUnit.

- Estamos usando Require.js? Como nosso exemplo é simples, não precisamos usá-lo.
- O Karma pode rodar os testes em diversos navegadores: Chrome, Chrome Canary, Firefox, Safari, Opera, IE, e PhantomJS. O PhantomJS, como citei, é um navegador *headless*, que roda na linha de comando, sem abrir uma janela. Caso seu projeto necessite, é possível rodar os testes em múltiplos navegadores (contanto que a máquina os tenha instalados).
- Devemos indicar a localização dos arquivos JavaScript a serem testados e a localização das specs.
- A configuração do Karma permite ignorar certos arquivos ao rodar os testes.
- Ao rodar a suite, o Karma pode monitorar modificações nos arquivos a serem testados ou nas specs e reexecutar os testes automaticamente.

Agora é só rodar a suite:

```
$ karma start karma.config.js
```

Você deverá ver algo assim:

```
INFO [karma]: Karma v0.12.1 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Mac OS X)]:
Connected on socket BXELksjC48w19PUuTdhW with id 98829501
PhantomJS 1.9.7 (Mac OS X): Executed 4 of 4 SUCCESS (0.005 secs /
0.003 secs)
```

O Karma continuará monitorando qualquer alteração em arquivos de teste ou implementação. Se adicionarmos um `console.log` em uma das specs, por exemplo, eis o que acontecerá:

```
INFO [watcher]: Changed file "/Users/eshiota/meuprojeto/spec/
formCartaoDeCreditoSpec.js".
LOG: {returnValue: true, timeStamp: 1395701896098, eventPhase: 0,
target: null, defaultPrevented: false, srcElement: null, type: 'submit',
clipboardData: undefined, cancelable: true, currentTarget: null,
bubbles: true, cancelBubble: false}
PhantomJS 1.9.7 (Mac OS X): Executed 4 of 4 SUCCESS (0.003 secs /
0.004 secs)
```

As vantagens de usar o Karma e o PhantomJS são:

- Não depender de um navegador com interface gráfica.
- Executar os testes automaticamente a cada alteração no código.
- Integrar o resultado dos testes com um servidor de integração contínua. O Karma fornece os resultados em um formato que pode ser integrado com diversas soluções.

11.5 Concluindo

Testes manuais podem parecer suficientes no começo, mas à medida que um projeto cresce, fica cada vez mais difícil garantir a integridade das funcionalidades. Com o uso cada vez maior no nosso dia a dia e com o aumento da complexidade do JavaScript, não podemos mais ignorar a importância de termos segurança para alterar um código e adicionar features sem impactar o funcionamento da aplicação. O problema toma dimensões ainda maiores quando diferentes desenvolvedores alteram a mesma base de código e potencialmente quebram funcionalidades.

Os testes unitários proporcionam essa segurança. Ao escrever specs, você garante que a implementação está fazendo exatamente o esperado, e pode testar se alterações, adições ou remoções no código alteraram o funcionamento da aplicação. Fazendo isso através do BDD ou TDD, você melhora o design da sua solução e adquire um conhecimento maior sobre o problema a ser resolvido. E, ao escrever uma nova spec a cada bug encontrado, você diminui a chance de causá-lo novamente.

Ferramentas para fazer isso não faltam, basta escolher a que mais lhe agradar. Neste artigo, utilizamos Jasmine e Karma, mas ambas são apenas duas entre as diversas opções disponíveis. E como vimos no final, é possível dar passos ainda maiores e integrar os testes de JavaScript em servidores de integração contínua, testando a integridade do código a cada versão gerada, ou em intervalos de tempo predeterminados.

O JavaScript não é mais apenas um artifício para fazer animações e enviar formulários. A linguagem mais utilizada no desenvolvimento para a web vai continuar crescendo, e é nosso papel como desenvolvedores garantir que nosso código e nossas aplicações sempre funcionarão.

Sobre o autor

Eduardo Shiota é desenvolvedor front-end e designer. Começou a produzir sites em 1998, trabalha com desenvolvimento para a Web desde 2002 e com design desde 2005. Usando HTML, CSS e JavaScript, busca projetar e implementar interfaces com boa usabilidade, através de um código limpo, semântico e escalável. Mora em Amsterdam desde 2013 com a esposa e seus quatro gatos; nas horas vagas, fotografa a cidade, e dá workshops, palestras e mentoria sobre front-end. Você pode segui-lo no Twitter como @shiota.